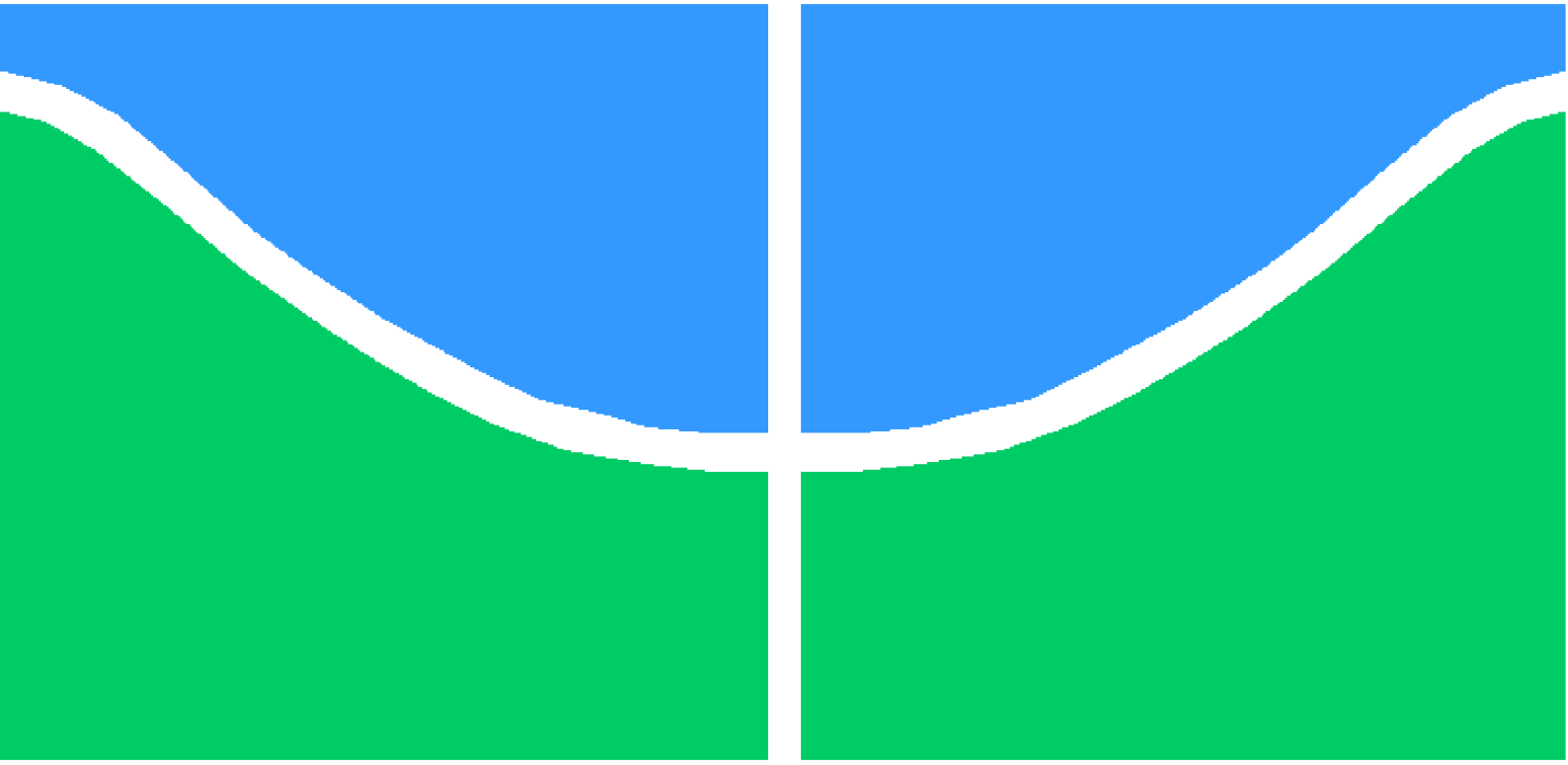


Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia Eletrônica

Implementação em FPGA de algoritmo de detecção de cantos de imagens para aplicações em tempo real

Autor: Matheus Bichara de Assumpção

Orientador: Professor Cristiano Jacques Miosso

Brasília, DF

2013

Matheus Bichara de Assumpção

Implementação em FPGA de algoritmo de detecção de cantos de imagens para aplicações em tempo real

Monografia submetida ao curso de graduação em Engenharia Eletrônica da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia Eletrônica.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Professor Cristiano Jacques Miosso

Brasília, DF

2013

Matheus Bichara de Assumpção

Implementação em FPGA de algoritmo de detecção de cantos de imagens para aplicações em tempo real/ Matheus Bichara de Assumpção. – Brasília, DF, 2013-
106 p. : il. (algumas color.) ; 30 cm.

Orientador: Professor Cristiano Jacques Miosso

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2013.

1. Processamento de imagens. 2. FPGA. I. Professor Cristiano Jacques Miosso. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Implementação em FPGA de algoritmo de detecção de cantos de imagens para aplicações em tempo real

CDU 02:141:005.6

Matheus Bichara de Assumpção

Implementação em FPGA de algoritmo de detecção de cantos de imagens para aplicações em tempo real

Monografia submetida ao curso de graduação em Engenharia Eletrônica da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia Eletrônica.

Trabalho aprovado. Brasília, DF, julho de 2013:

Professor Cristiano Jacques Miosso
Orientador

Professor Fabiano Araujo Soares
Convidado 1

Professor Daniel Mauricio Muñoz Arboleda
Convidado 2

Brasília, DF
2013

Agradecimentos

Agradeço aos meus pais, meu irmão e minha família que, com muito carinho e amor, estiveram presentes em mais esta etapa da minha vida, como meus maiores incentivadores.

Agradeço a todos os professores do curso, que foram tão importantes na minha vida acadêmica, em especial, ao Professor Cristiano Jacques Miosso, pela orientação e pelo estímulo que tornaram possível a conclusão desta monografia.

Agradeço aos amigos e colegas do curso, pelo convívio, pela compreensão e pela amizade.

Resumo

Os requerimentos de desempenho de aplicações de processamento de imagens têm aumentado continuamente a demanda por poder computacional, especialmente no que se refere a processamento em tempo real. Aplicações modernas como rastreamento, estimação de movimento, localização e mapeamento simultâneos (SLAM) e reconhecimento de objetos utilizam usualmente algoritmos de detecção de cantos como uma de suas primeiras etapas. É, portanto, essencial a execução desses algoritmos de forma eficiente. Nesse sentido, FPGAs proporcionam o desempenho do *hardware*, fazendo o uso de processamento paralelo, enquanto mantém a flexibilidade do *software*, com um custo relativamente baixo. A partir do referencial teórico levantado, este trabalho propõe e implementa uma arquitetura de *hardware* em dispositivo reconfigurável FPGA, para realizar a detecção de cantos de imagens em tempo real. Os resultados obtidos demonstraram a eficiência alcançada com a implementação proposta, que foi validada por meio de comparações com implementações em arquiteturas de computadores convencionais.

Palavras-chaves: Processamento de imagens, FPGA, *hardware* reconfigurável, detecção de cantos, VHDL.

Abstract

The performance requirements of image processing applications have been steadily increasing the demand for computational power, particularly in regard of real-time processing. Many applications such as feature tracking, motion estimation, simultaneous localization and mapping (SLAM) and object recognition usually have corner detection algorithms as one of its first steps. It is essential to perform these algorithms efficiently. Therefore, FPGAs provide hardware performance, with the advantages of parallel processing, while maintaining the flexibility of software, with a relatively low cost. This work proposes and implements an architecture for reconfigurable FPGA device to perform real-time corner detection in digital images. The results showed the efficiency achieved with the proposed implementation, which has been validated by comparisons with a conventional computer architecture implementation.

Key-words: Image processing, FPGA, Reconfigurable hardware, Corner detection, VHDL.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

1.1 Contextualização

Processamento de imagem engloba qualquer processo no qual as entradas e as saídas são imagens (GONZALEZ; WOODS, 2001), visando a sua melhoria e a extração de informações (ACHARYA; RAY, 2005).

Uma das primeiras aplicações de processamento de imagens remonta ao começo da década de 1920, onde buscavam-se formas de aprimorar a qualidade de impressão de imagens de jornais, transmitidas por cabo submarino, entre Londres e Nova Iorque (FILHO; NETO, 1999).

Entretanto, foi somente com o advento dos computadores digitais com poder de processamento e memória suficientes que o processamento digital de imagens expandiu-se (BAILEY, 2011). Um dos primeiros registros de processamento de imagens utilizando computador ocorreu em 1957, no qual foi feita a interface de um scanner a um computador no National Bureau of Standards, nos Estados Unidos (KIRSCH, 1998). Isso foi usado nos estágios iniciais de pesquisa em realce de bordas e reconhecimento de padrões.

Na década de 1960, a necessidade de processamento de uma enorme quantidade de imagens obtidas por satélites e explorações espaciais estimulou pesquisas no Jet Propulsion Laboratory na NASA (CASTLEMAN, 1996). Com o constante avanço dos computadores em poder de processamento e redução dos custos, houve um aumento considerável de aplicações para processamento digital de imagens, desde o controle de qualidade em processos industriais, geoprocessamento e meteorologia a pesquisas em astrologia (AIZAWA; SAKAUE; SUENAGA, 2004). Na área de Medicina, o uso de imagens tornou-se fundamental no diagnóstico médico, por meio de técnicas avançadas como a tomografia computadorizada e a ressonância magnética.

No entanto, aplicações atuais exigem, cada vez mais, maior complexidade computacional. Neste caso, o processamento por *hardware* especializado surge como um elemento decisivo, pois possibilita uma redução do tempo de processamento em decorrência da execução paralela das operações. Também, o advento da computação reconfigurável possibilita combinar o desempenho do *hardware* com a flexibilidade do *software*, permitindo o desenvolvimento de sistemas extremamente complexos (BAILEY, 2011).

O impulso na utilização de *hardware* reconfigurável só ocorreu na década de 1980, com o advento dos dispositivos programáveis, onde destacam-se os FPGAs (Field Programmable Gate Array). Uma arquitetura reconfigurável possibilita uma melhor adequação do *hardware* à aplicação, permitindo explorar estratégias diferentes em função da

aplicação a ser executada (SASS; SCHMIDT, 2010).

A computação reconfigurável proporciona uma maior flexibilidade nas arquiteturas dos computadores. Com a possibilidade do *hardware* também ser adaptável à aplicação, cria-se a possibilidade de se atingir um maior desempenho. Desse modo, um ponto importante é o desenvolvimento de uma arquitetura flexível, na qual o *hardware* seja implementado em lógica programável, utilizando os dispositivos como os FPGAs. Esses dispositivos são programados por meio de uma linguagem de descrição de *hardware*, como VHDL ou Verilog.

Em diversas aplicações nas áreas de visão computacional e processamento digital de imagens, a etapa de detecção de cantos é um componente essencial, como uma das primeiras etapas de processamento (Rosten; Drummond, 2006). Essa técnica envolve algoritmos de localização e descrição de pontos ou regiões de interesse em uma imagem, sendo muito utilizada em aplicações como: estimação de movimento, rastreamento, restauração de cena 3D, localização e mapeamento simultâneos (SLAM) e reconhecimento de objetos, entre outros.

Nesse contexto, o tema deste trabalho envolve a utilização de um dispositivo FPGA para a implementação de um algoritmo de detecção de cantos, mais especificamente o detector de cantos de Harris, que será detalhado durante o desenvolvimento.

1.2 Objetivos

1.2.1 Objetivo Geral

O objetivo geral deste trabalho é propor e implementar uma arquitetura em dispositivo reconfigurável FPGA, para realizar a detecção de cantos de imagens em tempo real, de forma a aproveitar a potencialidade do processamento paralelo e proporcionar um alto desempenho.

1.2.2 Objetivos Específicos

- Levantar o referencial teórico aplicado a imagens, detecção de cantos, e processamento de imagens em FPGA;
- Implementar algoritmo de detecção de cantos de Harris em *software* MATLAB;
- Propor uma solução paralela baseada em *hardware* para detectar cantos de imagens, utilizando o algoritmo de detecção de cantos de Harris;
- Descrever arquitetura projetada em linguagem VHDL e implementar-la em FPGA;
- Validar *design* em simulação e em FPGA;
- Realizar comparações de desempenho entre implementações em FPGA e em MATLAB.

1.3 Estrutura da dissertação

Esta dissertação encontra-se estruturada da seguinte forma: este primeiro capítulo trata da contextualização do tema, aponta o objetivo geral e os específicos do trabalho; no segundo capítulo, será apresentada a fundamentação teórica necessária ao desenvolvimento do projeto, sendo descritos os conceitos relacionados a processamento de imagens e dispositivos FPGA; a descrição da solução proposta será detalhada no terceiro capítulo; os resultados e análises serão apresentados no quarto capítulo; por fim, no quinto capítulo, serão apresentadas as conclusões deste trabalho.

2 Fundamentação teórica

Neste capítulo, os principais conceitos teóricos necessários ao desenvolvimento deste trabalho serão estudados. Os seguintes temas serão abordados: processamento de imagens digitais e *hardware* reconfigurável/FPGA.

Na primeira seção, na qual será tratado o processamento de imagens digitais, serão apresentados conceitos de imagens digitais, amostragem e quantização, vizinhança de um pixel e operadores locais, filtragem linear de imagens, detecção de bordas e características. Além disso, o algoritmo de detecção de cantos de Harris será abordado com propriedade.

Na segunda seção, apresentam-se os conceitos relacionados a *hardware* reconfigurável, dispositivos FPGA e linguagens de descrição de *hardware* e VHDL.

Por fim, o uso de FPGA em processamento de imagens será contextualizado, assim como o conceito de processamento multicamada em *hardware*.

2.1 Processamento digital de imagens

2.1.1 Imagem digital

Uma imagem pode ser definida como uma função bidimensional, $f(x,y)$, onde x e y são coordenadas espaciais e a amplitude f, em um par de coordenadas (x,y) qualquer, é chamada de intensidade ou nível de cinza da imagem nesse ponto (GONZALEZ; WOODS, 2001).

Tendo valores discretos e finitos para x, y e f, tem-se a imagem digital.

Uma imagem digital pode ser entendida, então, como sendo uma matriz (vetor bidimensional) de elementos. Esses elementos são chamados pixels. O pixel é o elemento básico de uma imagem, podendo ser entendido como o valor proporcional ao brilho no ponto correspondente na cena (Nixon; Aguado, 2008). De acordo com (GONZALEZ; WOODS, 2001), uma imagem digital, $f(x,y)$, de tamanho M x N, pode ser escrita de acordo com a matriz mostrada na Eq. (2.1).

$$f(x,y) = \begin{bmatrix} f(0,0) & f(0,1) & \cdots & f(0,N-1) \\ f(1,0) & f(1,1) & \cdots & f(1,N-1) \\ \vdots & \vdots & \ddots & \cdots \\ f(M-1,0) & f(M-1,1) & \cdots & f(M-1,N-1) \end{bmatrix} \tag{2.1}$$

A Fig. (1) mostra o exemplo de uma imagem digital de um carro, onde o retrovisor

é representado por uma matriz de números.

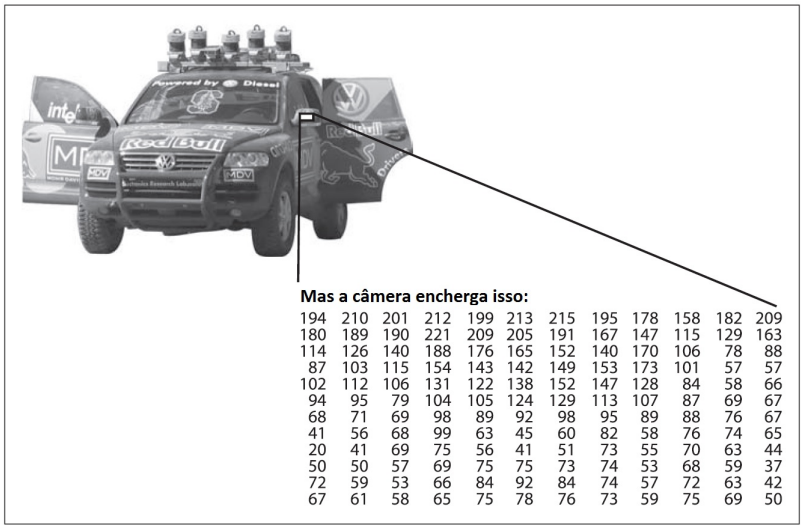


Figura 1 – O retrovisor de um carro é representado por uma matriz de números em uma imagem digital. Adaptado de (BRADSKI; KAEHLER, 2008)

Para converter uma cena real em uma imagem digitalizada, duas etapas são imprescindíveis: a aquisição da imagem e sua digitalização (FILHO; NETO, 1999).

A aquisição é o processo de conversão de uma cena tridimensional em uma imagem analógica. Um dos dispositivos de aquisição de imagens muito utilizado atualmente é a câmera CCD(Charge Coupled Device), que consiste em uma matriz de células semicondutoras fotossensíveis, que atuam como capacitores, armazenando carga elétrica proporcional à energia luminosa incidente (FILHO; NETO, 1999).

O sinal resultante é então condicionado por circuitos eletrônicos especializados, produzindo um Sinal Composto de Vídeo (SCV), analógico e monocromático.

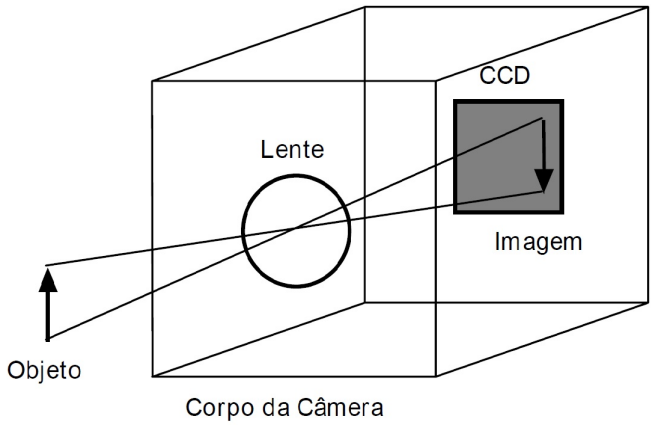


Figura 2 – Visão esquemática de uma câmera CCD. Retirado de (FILHO; NETO, 1999)

Uma câmera CCD monocromática simples consiste basicamente de um conjunto de lentes que focalizarão a imagem sobre a área fotossensível do CCD, o sensor CCD e seus

circuitos complementares (FILHO; NETO, 1999). A Fig. (2) demonstra simplificadamente a aquisição de imagens com câmera CCD.

A etapa de digitalização é feita com base nos conceitos de amostragem e quantização que serão descritos a seguir.

2.1.2 Amostragem e quantização

Segundo (ACHARYA; RAY, 2005), entender os processos de amostragem e quantização é um dos pontos fundamentais em processamento de imagens.

Para que uma imagem possa ser tratada computacionalmente, faz-se necessário, como mencionado na seção anterior, que a imagem seja digitalizada. E para isso, faz-se necessária a realização de uma amostragem e quantização da imagem analógica obtida por sensores.

A amostragem pode ser entendida como a discretização no espaço, enquanto a quantização se refere à discretização em amplitude, de acordo com (GONZALEZ; WOODS, 2001).

O processo de amostragem consiste na conversão da imagem analógica em uma matriz com M x N pontos (pixels), mostrada anteriormente na Eq. (2.1). A amostragem mais comum e mais popular é a chamada de uniformemente espaçada, onde cada amostra é tomada em intervalos iguais. Existem outras técnicas de amostragem que utilizam espaçamento de tamanhos direfentes, sendo menos comuns.

Os valores M e N definem a resolução da imagem. Maiores valores implicam em uma maior resolução, que está diretamente relacionada à qualidade subjetiva da imagem. A qualidade de imagem é um conceito altamente subjetivo, que também depende fortemente dos requisitos da aplicação dada (FILHO; NETO, 1999). Uma maior resolução implica, porém, em uma maior complexidade computacional, necessária para realizar tratamentos na imagem, bem como um maior custo de armazenamento (FILHO; NETO, 1999).

A Fig. (3) apresenta uma mesma imagem com diferentes resoluções. Da esquerda para direita, tem-se as resoluções de 128 x 128, 64 x 64 e 32 x 32, respectivamente. Verifica-se que, com o aumento da resolução, obtém-se contornos mais definidos.

Na quantização, por sua vez, cada valor de pixel assume um valor inteiro, na faixa de 0 a 2^n -1, onde n é o número de tons de cinza. A imagem resultante terá então 2^n possíveis tons de cinza. Assim, n é o número de bits necessários para representar cada pixel.

A quantização mais comum consiste em tomar o valor máximo e o valor mínimo dos pixels da imagem, e dividir este segmento em intervalos iguais de acordo com o número

Figura 3 – Uma mesma imagem com diferentes resoluções. Da esquerda para direita, 128 x 128, 64 x 64 e 32 x 32. Adaptado de (GONZALEZ; WOODS, 2001)

de bits definido para armazenar uma amostra. Comumente, utiliza-se o valor de n igual a 8, o que equivale a um byte. Isso representa uma variação de 0 a 255 nos tons da imagem.

Na Fig. (4) é possível se comparar uma mesma imagem com diferentes quantizações, o que ilustra os efeitos da redução do número de níveis de cinza sobre a qualidade da imagem. Quanto menor o número de níveis de cinza, é mais perceptível o surgimento de uma imperfeição na imagem, conhecida como falso contorno (FILHO; NETO, 1999).

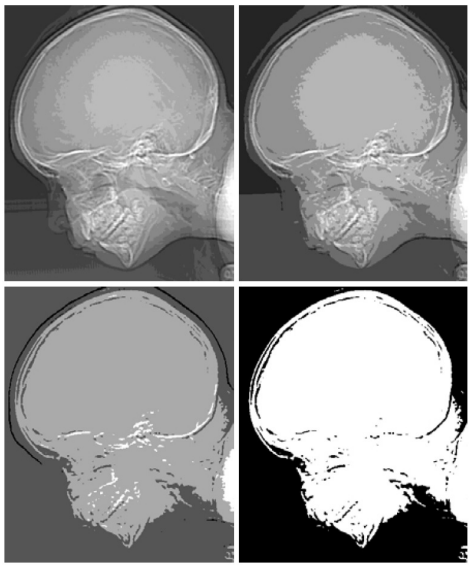


Figura 4 – Uma mesma imagem com diferentes níves de cinza. Da esquerda para direita, de cima para baixo: 16, 8, 4 e 2 níveis de cinza. Fonte: (GONZALEZ; WOODS, 2001)

2.1.3 Vizinhança de um pixel

Um pixel qualquer p nas coordenadas (x, y) possui dois pixels vizinhos horizontais (um à esquerda e outro à direita) e outros dois verticais (um acima e outro abaixo). Esse conjunto é chamado de vizinhança de 4 de p (JAIN, 1989), representada também por $N_4(p)$.

De modo similar, para um pixel p existem quatro vizinhos diagonais, denominados $N_d(p)$.

Quando os pixels da vizinhança diagonal são combinados com os da vizinhança de 4, se obtém a vizinhança de 8 de p, também representada por $N_8(p)$. Na Fig. (5) podem ser visualizados os conceitos de vizinhança.

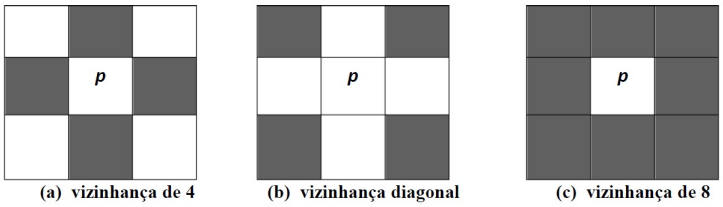


Figura 5 – Relações de vizinhança entre pixels

Para algumas regiões de uma imagem, a vizinhança pode possuir algum pixel fora dos domínios da imagem se o ponto p estiver localizado na borda da imagem.

É necessária a compreensão dos conceitos de vizinhança para o entendimento de técnicas de processamento de imagens baseadas nas relações entre os pixels. Técnicas de processamento em vizinhança são, por exemplo, peças fundamentais para processos de realce e restauração de imagens. Algumas vantagens de se realizar processamento em vizinhança são a velocidade operacional e a simplicidade para implementação em *hardware* (ANNADURAI, 2007).

2.1.4 Operadores locais e de vizinhança

Os operadores locais e operadores de vizinhança são extensamente utilizados em processamento de imagens. Esses operadores realizam operações no chamado domínio espacial.

O termo domínio espacial se refere ao agregado de pixels que compõem uma imagem (ACHARYA; RAY, 2005). Métodos em domínio espacial operam diretamente nesses pixels. Processos no domínio espacial são normalmente representados como se observa na Eq. (2.2), onde *f(x,y)* é a imagem de entrada, *g(x,y)* é a imagem processada, e T é um operador em *f*, definido sobre alguma vizinhaça de *(x,y)*.

$$g(x, y) = T[f(x, y)] \tag{2.2}$$

A principal abordagem em definir uma vizinhança ao redor do ponto *(x,y)* é usar uma subimagem de área quadrada ou retangular, com centro em *(x,y)*. O centro dessa subimagem é movido pixel por pixel, começando, usualmente, no canto superior esquerdo da imagem.

O operador T é aplicado em cada posição *(x,y)* para resultar na saída, g, em cada localidade.

A forma mais simples de T é quando tem-se uma vizinhança de tamanho 1 x 1, ou seja, um único pixel. Uma operação desse tipo bastante utilizada é a limiarização ou *thresholding*, mostrada na Fig.(6). Considerando uma imagem em escala de cinza, valores de r acima de m possuem um mesmo valor, bem como valores abaixos têm o valor zero. Esse operador $T(r)$ produz então uma imagem binária. Operações desse tipo são geralmente chamadas de ponto a ponto.

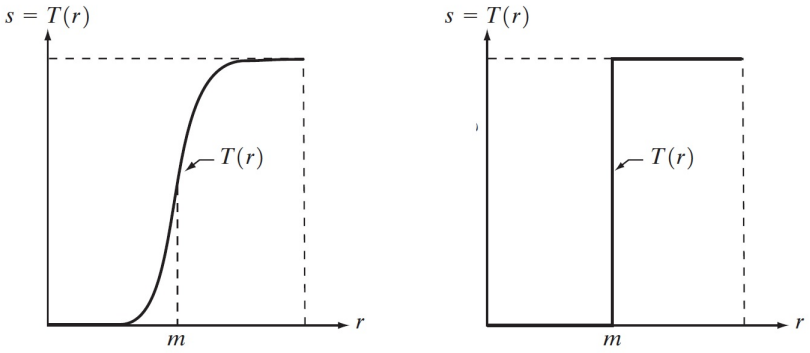


Figura 6 – Exemplo de operação de limiarização em imagens. Fonte: (GONZALEZ; WOODS, 2001)

Maiores vizinhanças proporcionam consideravelmente mais flexibilidade (GONZALEZ; WOODS, 2001). Uma das principais abordagens é a utilização das chamadas máscaras, filtros, kernels ou janelas. De modo simples, uma máscara é um pequeno vetor 2D, por exemplo 3 x 3.

Os valores dos coeficientes da máscara determinam qual é sua operação, como, por exemplo, o aguçamento de imagens, comumente encontrado em realce. Processamentos desse tipo são geralmente referidos como processamento de máscara ou filtragem, os quais serão discutidos na próxima seção.

2.1.5 Filtragem linear no domínio espacial

Como já mencionado, as operações em vizinhança envolvem os valores dos pixels da imagem em uma vizinhança e os valores correspondentes de uma subimagem que tem as mesmas dimensões que a vizinhança. Essa subimagem é chamada de filtro, máscara, kernel ou janela, enquanto os valores em um filtro são chamados de coeficientes, ao invés de pixels (GONZALEZ; WOODS, 2001).

O conceito de filtragem tem suas origens no uso da transformada de Fourier para processamento de sinais, no chamado domínio da frequência.

Uma abordagem muito comum em filtragem é a realização de operações diretamente nos pixels da imagem. Esse processo é chamado de filtragem no domínio espacial, para diferenciar esse tipo de processo da filtragem no domínio da frequência.

O funcionamento da filtragem espacial é ilustrada na Fig.(7). O processo consiste em simplesmente mover a máscara de ponto a ponto na imagem.

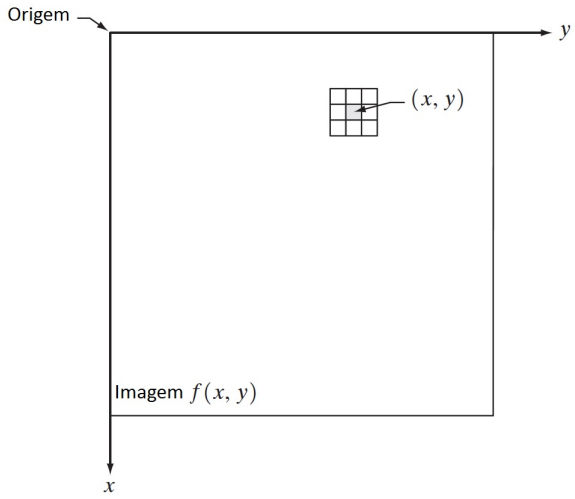


Figura 7 – Uma vizinhança 3 x 3 ao redor de um ponto *(x,y)* em uma imagem. Fonte: (GONZALEZ; WOODS, 2001)

Em cada ponto *(x,y)*, a resposta do filtro é calculada por meio de uma determinada relação. A Fig.(8) ilustra esse processo.

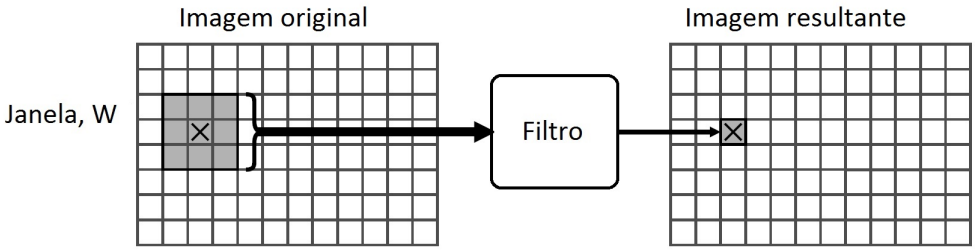


Figura 8 – Aplicando um filtro em uma vizinhança 3 x 3 ao redor de um ponto (x,y) em uma imagem. Fonte: (BAILEY, 2011)

Para uma filtragem linear, a resposta é dada pela soma dos produtos dos coeficientes do filtro e os pixels da subimagem correspondente. São usualmente utilizadas máscaras de tamanho ímpar, a partir de 3 x 3, pois 1 x 1 implicaria uma máscara de um coeficiente apenas. Então, para uma máscara de tamanho m x n, assume-se que $m = 2a + 1$ e $n = 2b + 1$, onde a e b são inteiros não negativos.

A filtragem linear de uma imagem f de tamanho M x N com um filtro w de tamanho m x n é dada pela Eq. (2.3), onde $a = (m - 2)/2$ e $b = (n - 1)/2$.

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x+s, y+t) \tag{2.3}$$

Para gerar uma imagem completamente filtrada, essa equação deve ser aplicada para $x = 0, 1, 2, ..., M - 1$ e $y = 0, 1, 2,..., N - 1$. Dessa forma, garante-se que todos os pixels da imagem serão processados.

O processo de filtragem linear no domínio espacial é usualmente chamado de convolução. Por isso, filtros são algumas vezes denotados por máscara de convolução ou kernel de convolução. Na convolução, porém, antes de ser realizada a operação da Eq. (2.3), chamada de correlação, o kernel é rebatido horizontalmente e verticalmente, ou espelhado.

A grande diferença entre a convolução e a correlação é o fato de a convolução ser associativa. Isso é, se F e G são filtros, então $F * (G * I)$ é igual a $(F * G) * I$. A convolução e a correlação são, portanto, iguais quando o filtro for simétrico.

É muito conveniente o fato de a convolução ser associativa. Por exemplo, se houver o desejo de se realizar a suavização de uma imagem e, logo após, encontrar suas derivadas, o mais intuitivo seria a aplicação de duas filtragens consecutivas. É possível, no entanto, realizar-se somente uma operação de filtragem, com o filtro resultante da convolução dos dois anteriores. Isso é vantajoso porque o filtro pode ser pré-calculado, reduzindo a quantidade de cálculos necessários.

Uma importante consideração ao se implementar operações em vizinhança é como se proceder quando o centro do filtro se aproxima da borda da imagem. Se o centro da máscara estiver perto da borda, possivelmente uma ou mais linhas ou colunas estarão localizadas fora da imagem. Existem algumas possibilidades para se lidar com esse problema.

Considerando um kernel de tamanho n x n, a maneira mais simples de se resolver a questão é garantir que o centro da máscara esteja a uma distância de não menos de $(n - 1)/2$ da borda. A imagem resultante será então menor que a imagem original, mas todos os pixels terão sido processados pela máscara completa.

Se for necessário que a imagem resultante tenha o mesmo tamanho que a imagem original, uma possibilidade é processar os pixels perto da borda somente com uma parte da máscara, que se encontra na imagem. Assim, alguns pixels na borda terão sido processados somente com uma parte do kernel.

Outra opção é realizar o *padding* da imagem, onde se adicionam linhas e colunas de zeros ou outras constantes nas bordas, e efetuar o processamento normalmente. Desse modo, mantém-se o tamanho da imagem original. Entretanto, essa abordagem causa o chamado efeito de borda na imagem resultante. A única forma de se obter um resultado perfeitamente filtrado é aceitar uma imagem ligeiramente menor limitando o centro do filtro a uma distância não menor que $(n - 1)/2$ pixels da borda (GONZALEZ; WOODS, 2001).

2.1.6 Detecção de bordas

A detecção de bordas é um dos processos mais comuns na análise de imagens digitais. Refere-se ao processo de identificar e localizar descontinuidades em uma imagem (MAINI; AGGARWAL, 2009)e é frequentemente o primeiro passo visando à obtenção de informações nas imagens a serem tratadas (Nixon; Aguado, 2008). As bordas ou contornos são variações bruscas na intensidade dos pixels, que normalmente caracterizam os contornos dos objetos na cena. Existe uma borda onde ocorre mudanças, por exemplo, de pixels com valores baixo (preto) para pixels com valores alto (branco), ou reciprocamente, de pixels com valores alto (branco) para pixels com valores baixo (preto). Tipicamente, as bordas estão localizadas nas fronteiras entre duas ou mais regiões na imagem, destacando cenas ou objetos.

A Fig. (9) ilustra a detecção de bordas em uma imagem e mostra a imagem original e a resultante após o processamento.

Figura 9 – Aplicação de detecção de bordas a uma imagem. Fonte: (Nixon; Aguado, 2008)

Percebe-se que, mesmo não contando com todas as informações da imagem original, é possível se obter uma boa compreensão da cena, observando-se somente a imagem resultante. A detecção de bordas, portanto, mantém informações de contorno da imagem original, mesmo com a redução da quantidade de dados, o que a torna vantajosa em diversas aplicações em processamento de imagens e visão computacional em geral.

Uma borda não é resultante somente da geometria dos objetos da cena, como se pensa geralmente. As seguintes descontinuidades também podem ser consideradas como bordas:

- Descontinuidades em profundidade;
- Descontinuidades na orientação de uma superfície;
- Mudanças nas propriedades de um material;

- Variações na iluminação de uma cena;

Muitos algoritmos de detecção de bordas já foram propostos. Esses algoritmos diferem em aspectos como: custo computacional, desempenho e facilidade de implementação em *hardware* (ALZAHRANI; CHEN, 1997). A detecção de bordas é realizada em processamento digital de imagens normalmente por meio de operadores diferenciais de primeira e segunda ordem (GONZALEZ; WOODS, 2001). A Fig. (10) mostra as transições de nível de cinza de uma linha de uma imagem, e sua primeira e segunda derivada.

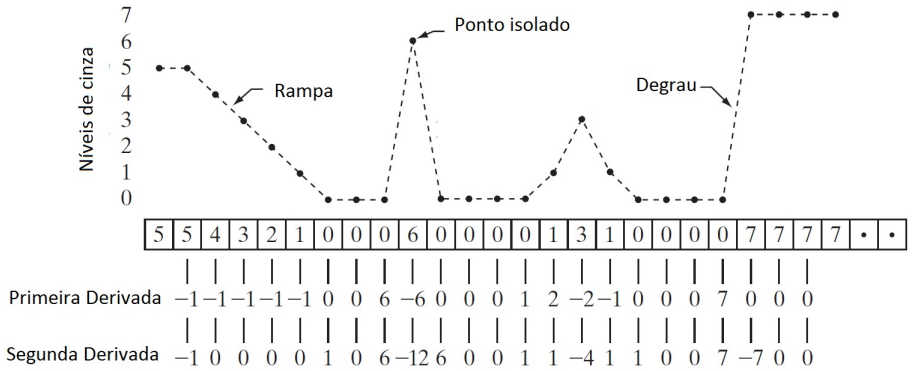


Figura 10 – Transições de níveis de cinza e primeira e segunda derivadas. Adaptado de: (GONZALEZ; WOODS, 2001)

A primeira derivada é constante e positiva nos pontos de transição e zero para áreas com valores de cinza constantes. A segunda derivada, por sua vez, é positiva na região associada à parte mais escura da borda, negativa na região associada à área mais clara, e zero ao longo da rampa e área de intensisade de cinza constantes. É possível concluir com essas observações que a magnitude da primeira derivada pode ser usada para detectar a presença de uma borda em um ponto da imagem. De maneira similar, o sinal da segunda derivada pode ser utilizado para determinar se um pixel da borda está no lado escuro ou claro dela. Percebe-se que, imaginando-se uma linha reta cruzando os valores positivo e negativo da segunda derivada, se atingiria zero próximo à metade da borda. Essa propriedade é útil para localizar o centro de bordas largas.

Operadores de segunda ordem possuem, todavia, a desvantagem de serem muito sensíveis a ruído (FILHO; NETO, 1999). As derivadas de primeira ordem em uma imagem são calculadas usando o Gradiente, enquanto derivadas de segunda ordem são obtidas utilizando-se o Laplaciano. Tanto o Gradiente quanto o Laplaciano costumam ser aproximados por kernels (máscaras) de convolução ou operadores 3 x 3. Alguns exemplos destes kernels são mostrados na Tab. (1).

O Gradiente de uma imagem $f(x,y)$ no ponto (x,y) é definido como o vetor mostrado

Operador	Vertical	Horizontal
Roberts	$\begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$
Sobel	$\frac{1}{4}\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$	$\frac{1}{4}\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$
Prewitt	$\frac{1}{3}\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$	$\frac{1}{3}\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$
Frei-Chen	$\frac{1}{2+\sqrt{2}}\begin{bmatrix} 1 & 0 & -1 \\ \sqrt{2} & 0 & -\sqrt{2} \\ 1 & 0 & -1 \end{bmatrix}$	$\frac{1}{2+\sqrt{2}}\begin{bmatrix} -1 & -\sqrt{2} & -1 \\ 0 & 0 & 0 \\ 1 & \sqrt{2} & 1 \end{bmatrix}$

Tabela 1 – Operadores de primeira ordem para detecção de bordas. Fonte: (FILHO; NETO, 1999)

na Eq. (2.4).

$$\nabla \mathbf{f} = \begin{bmatrix} G_x \\ G_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} \tag{2.4}$$

O vetor gradiente aponta para a região de maior variação de f nas coordenadas (x,y). A magnitude desse vetor, bem como a sua direção, é de grande importância em detecção de bordas, e pode ser definida pela Eq. (2.5).

$$mag(\nabla \mathbf{f}) = \sqrt{(G_x^2 + G_y^2)} \tag{2.5}$$

Existem diversos operadores de gradiente para detecção de bordas. Entre eles destaca-se o operador de Sobel, bastante eficiente, que possui uma implementação computacional simples. Podem-se destacar também os operadores de Prewitt e Roberts.

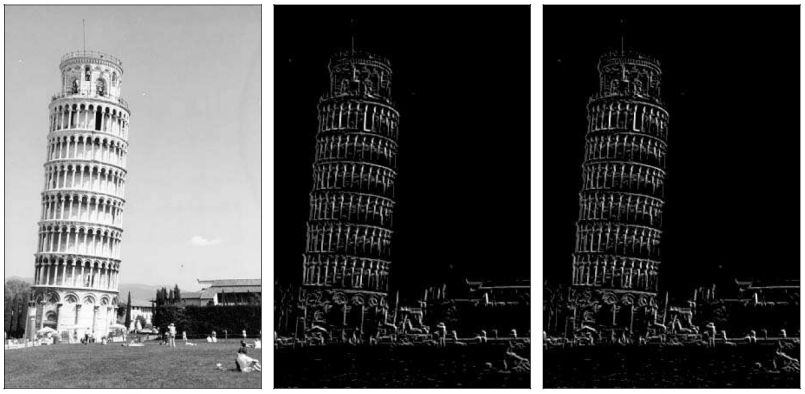
A Fig. (11) mostra algumas operações de detecção de bordas com diferentes kernels.

2.1.7 Detecção de cantos

Cantos em imagens representam uma grande quantidade de informação. Detectar cantos de forma precisa é bastante significativo em processamento de imagens e visão computacional, o que pode reduzir muito a quantidade de cálculos (CHEN et al., 2009).

Cantos são características locais importantes em imagens. São pontos de alta curvatura e se localizam na junção de diferentes regiões de brilho de imagens, possuindo bordas em duas ou mais direções (Nixon; Aguado, 2008).

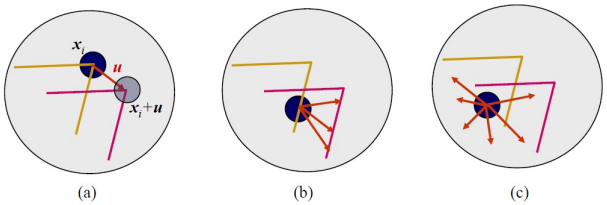
Intuitivamente, um canto pode ser entendido como um ponto distintivo numa imagem, que é mais susceptível de ser encontrado em uma outra imagem. Por exemplo,



(a) (b) (c)

Figura 11 – Exemplo de realce e detecção de bordas. (a) imagem original, (b) realce de bordas utilizando os operadores de Prewitt horizontal e vertical, (c) realce de bordas utilizando os operadores de Sobel horizontal e vertical. Fonte: (FILHO; NETO, 1999)

tendo duas cenas diferentes, ao escolher uma característica numa imagem não seria muito bom se escolher uma parede ou outra área plana, porque não há possibilidade de dizer com certeza em que parte na parede em uma outra imagem essa característica pode ser encontrada. Haverá quase o mesmo problema escolhendo um ponto em uma aresta. Provavelmente será reconhecida uma borda em outra imagem, mas não será possível de dizer onde exatamente a característica escolhida está localizada. A solução é escolher uma característica com propriedades únicas que não possibilite redundância: um canto. A Fig. (12) exemplifica essas idéias.



(a) (b) (c)

Figura 12 – (a) Canto (b) Aresta (b) Área plana. Fonte: (SZELISKI, 2010)

Uma aplicação interessante é a construção de imagens panorâmicas. Tendo-se como base as duas imagens, pode-se construir uma única imagem por meio do alinhamento das duas. Ver Fig. (13).

Tendo duas imagens, o primeiro passo para fazer o alinhamento é detectar pontos característicos, ou cantos, em cada imagem.

O segundo passo é encontrar os pares correspondentes de regiões entre as duas imagens.

O terceiro e último passo é então usar os pares encontrados para efetivamente alinhar as imagens e gerar a imagem panorâmica.

Figura 13 – Exemplo de formação de imagem panorâmica. Da esquerda para a direita, de cima para baixo: imagens originais; detecção de cantos em ambas imagens; correspondência entre regiões; alinhamento de imagens. Fonte: (SZELISKI, 2010)

Outras aplicações da detecção de cantos inclui reconstrução 3D, rastreamento de movimento, reconhecimento de objetos, navegação de robôs e outros.

Vários algoritmos de detecção de cantos já foram propostos por pesquisadores. Desde que os primeiros detectores de cantos foram desenvolvidos no final da década de 1970, dezenas de outros detectores foram propostos. A Fig. (14) mostra a linha do tempo dos detectores desenvolvidos até o final da década de 1990.

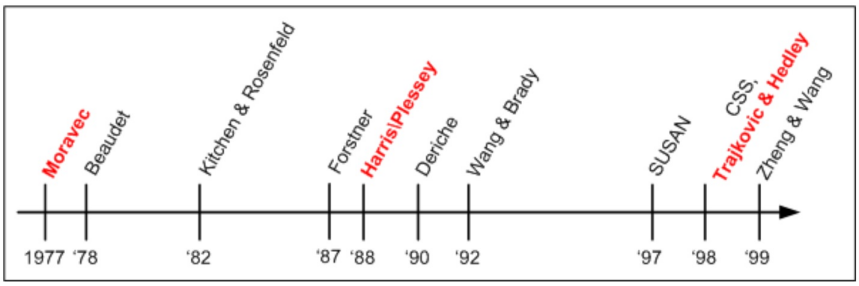


Figura 14 – Avanço em algoritmos de detecção de cantos até o final da década de 1990. Fonte: (D.PARKS; GRAVEL,)

Existem diversas abordagens na literatura para se realizar a detecção de cantos. Um dos algoritmos mais populares é o detector de cantos de Harris (HARRIS; STEPHENS, 1988), que faz parte da mesma família de detectores destacados em vermelho na Fig. (14). Suas principais características, de acordo com (JIAO; HE, 2009), são:

- O operador é relativamente simples e é adequado para detecção automática de características locais;
- Os pontos detectados são bem proporcionados e válidos;

- A quantidade de pontos detectados é determinada de acordo com o requerimento do usuário;
- As características locais são invariantes à translação e à rotação, e o operador é estável.

Esse algoritmo será explicado detalhadamente na próxima seção.

2.1.8 Detector de cantos de Harris

O algoritmo de detecção de cantos de Harris se baseia nos pensamentos de (MORAVEC, 1979). Tendo-se uma janela com centro no pixel $p(x,y)$ e movendo-a na vizinhança de p, as variações podem ser mensuradas de acordo com a função de autocorrelação, descrita na Eq. (2.6), onde (x_k,y_k) são pontos dentro da janela W com centro no ponto $p(x,y)$.

$$f(x,y) = \sum_{x_k,y_k \in W} (I(x_k + \Delta x, y_k + \Delta y) - I(x_k, y_k))^2 \tag{2.6}$$

(MORAVEC, 1979) cita que as variações se comportam da seguinte maneira:

- Pequenas variações ocorrem em todas as direções quando a vizinhança tem uma intensidade constante, representando, por exemplo, uma parede, chão etc;
- Pequenas variações ocorrem em somente uma direção quando há uma borda;
- Grandes variações em todas as direções indicam a presença de um canto;

A Fig. (15) ilustra esses três casos.

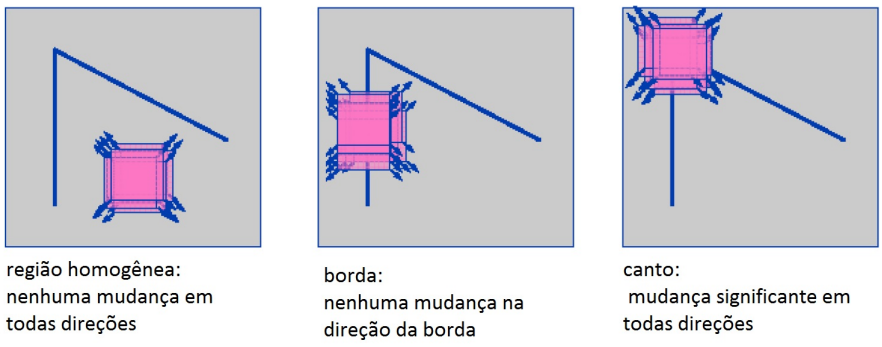


Figura 15 – Região homogênea, borda e canto. Fonte: (COLLINS, 2005)

Esse algoritmo, entretanto, não é eficiente do ponto de vista computacional, pelo fato de a janela ser movida em todas as direções, e para cada movimento as variações terem que ser calculadas.

(HARRIS; STEPHENS, 1988) introduziu então o conceito da matriz de autocorrelação, que é derivada da Eq. (2.6). Inicialmente, realiza-se a expansão de Taylor para o termo $I(x_k + \Delta x, y_k + \Delta y)$, como visto na Eq. (2.7).

$$I(x_k + \Delta x, y_k + \Delta y) \approx I(x_k, y_k) + \begin{pmatrix} I_x(x_k, y_k) & I_y(x_k, y_k) \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} \tag{2.7}$$

Na expansão de Taylor, I_x e I_y são as derivadas da imagem nas correspondentes direções, geralmente computadas utilizando-se a máscara de Sobel. Realizando-se a combinação das Eq. (2.6) e Eq. (2.7) é obtida a Eq. (2.8), onde G' é chamada de matriz de autocorrelação.

$$\begin{aligned} f(x,y) &= \sum_{x_k, y_k \in W} \left(\begin{pmatrix} I_x(x_k, y_k) & I_y(x_k, y_k) \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} \right)^2 \\ &= \begin{pmatrix} \Delta x & \Delta y \end{pmatrix} \begin{bmatrix} \sum_{x_k, y_k \in W} I_x^2 & \sum_{x_k, y_k \in W} I_x I_y \\ \sum_{x_k, y_k \in W} I_x I_y & \sum_{x_k, y_k \in W} I_y^2 \end{bmatrix} \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} \\ &= \begin{pmatrix} \Delta x & \Delta y \end{pmatrix} G'(x,y) \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} \end{aligned} \tag{2.8}$$

Para reduzir o erro de localização do canto de Harris, é recomendado se utilizar uma janela do tipo gaussiana. O resultado é a matriz de autocorrelação ponderada, na Eq. (2.9).

$$\begin{aligned} G(x,y) &= \begin{bmatrix} \sum_{x_k, y_k \in W} w_k I_x^2 & \sum_{x_k, y_k \in W} w_k I_x I_y \\ \sum_{x_k, y_k \in W} w_k I_x I_y & \sum_{x_k, y_k \in W} w_k I_y^2 \end{bmatrix} \\ &= \begin{bmatrix} \sum_{x_k, y_k \in W} G_{x2} & \sum_{x_k, y_k \in W} G_{xy} \\ \sum_{x_k, y_k \in W} G_{xy} & \sum_{x_k, y_k \in W} G_{y2} \end{bmatrix} \end{aligned} \tag{2.9}$$

Os autovalores λ_1 e λ_2 dessa matriz descrevem as variações dentro da janela de modo similar à janela descrita por (MORAVEC, 1979), da seguinte forma:

- Pequenos autovalores de *G(x,y)* correspondem a pequenas mudanças em todas as direções, ou seja, uma textura plana;
- Um autovalor significa que existe uma borda dentro da janela;

- Um canto é descrito por dois autovalores elevados;

Assim, o espaço de autovalores pode ser dividido entre plano(*flat*), borda(*edge*) e canto(*corner*), como mostrado na Fig (16).

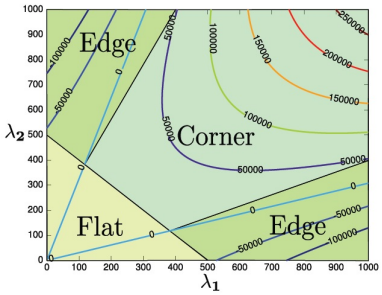


Figura 16 – Autovalores da matriz de autocorrelação para determinação de cantos. Fonte: (HARRIS; STEPHENS, 1988)

Como o cálculo desses autovalores é computacionalmente muito intenso, Harris desenvolveu uma medida para classificação, denominado aqui por fator de Harris.

Essa medida é dada pela Eq. (2.10), onde k é um valor determinado empiricamente, com valores recomendados entre 0.03 e 0.08, sendo 0.04 o valor ótimo.

$$\begin{aligned} R &= Det(G(x,y)) - k.trace^2(G(x,y)) \\ &= (G_{x2}G_{y2} - G_{xy}^2) - k(G_{x2} + G_{y2})^2 \end{aligned} \tag{2.10}$$

Para determinar efetivamente os cantos, realiza-se uma operação de limiarização (*threshold*) sobre a resposta do fator de Harris. É normalmente feita a supressão dos valores não máximos em uma vizinhança. A Fig. (17) mostra a detecção de cantos de uma imagem.

2.2 *Hardware* Reconfigurável

2.2.1 Dispositivos de lógica programável

Computadores convencionais são baseados na utilização de uma Unidade Lógica Aritmética (ULA), que pode executar uma de várias operações, baseadas em um conjunto de sinais de controle. A ULA possui a limitação, porém, de realizar somente uma operação por vez. Por isso, para aplicações mais complexas, deve-se quebrar o procedimento em várias operações básicas, seguindo uma sequência de sinais de controle armazenada em memória.

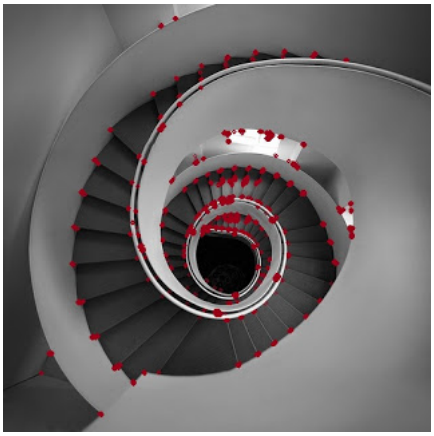


Figura 17 – Cantos encontrados ao se utilizar o detector de cantos de Harris.

A ideia de se ter um *hardware* programável é desenvolver um circuito genérico onde a funcionalidade pode ser programada para uma aplicação específica. Dispositivos simples de lógica programável começaram a aparecer no início da década de 1970, e eram usados principalmente para simplificar circuitos contendo diversos dispositivos de lógica discreta. Esses dispositivos eram programados por fusíveis ou máscaras, onde alterava-se a sua camada de metal durante a fabricação, sendo possível programá-los uma única vez (BAILEY, 2011).

Os dispositivos lógico-programáveis (PLDs) surgiram como uma nova opção para satisfazer a demanda de flexibilidade nos circuitos lógicos digitais. A habilidade de adequar-se às diferentes aplicações com necessidades de estruturas computacionais e de comunicação específicas foi um dos grandes atrativos desse novo segmento (BAILEY, 2011).

Um grande avanço, na metade da década de 1980, foi o controle de cada conexão programável como uma célula EEPROM, ao invés de um fusível. Isso possibilitou os dipositivos serem apagados e reprogramados. Com isso, a lógica pôde ser alterada simplesmente reprogramando o dispositivo. Em 1985, foram introduzidos no mercado os FPGAs, pela fabricante Xilinx, baseados na arquitetura LUT (lookup table) (NAVABI, 2005).

Lógica programável representa a funcionalidade como um circuito, onde um circuito específico pode ser programado para atender os requisitos de uma aplicação. A grande diferença de um sistema computacional tradicional é que a funcionalidade é implementada em um sistema paralelo, em vez de sequencial.

2.2.2 FPGA

Um Field-Programmable Gate Array (FPGA) é um circuito integrado digital que possui um conjunto de blocos lógicos organizados em forma de matriz e respectivas interligações, que podem ser configurados de modo a criar um circuito digital arbitrário (WOODS et al., 2008). Um FPGA pode implementar desde uma simples lógica combinacional,

até blocos de memórias e estruturas complexas, como um processador. A estrutura de conexão é capaz de rotear cada bloco lógico de forma a conectá-los da melhor maneira possível para garantir o funcionamento do circuito implementado.

A estrutura básica do FPGA (Fig. (18)) pode variar segundo o fabricante do mesmo, mas é composta, basicamente, dos seguintes recursos:

- Funções lógicas programáveis de n entradas (blocos lógicos), onde n varia com a família e fabricante do FPGA;
- Rede de conexão para interligar entre os diversos blocos lógicos existentes;
- Flip-flops ou registradores (blocos lógicos) para o armazenamento de informações;
- Amplificadores de corrente de entrada e saída;
- Memória RAM interna nos dispositivos mais modernos;

FPGAs que possuem um pequeno número de poderosos blocos lógicos reconfiguráveis são classificados como FPGAs com granulações grandes, enquanto que os que possuem grande número de blocos lógicos simples, são classificados como FPGAs com granulações pequenas [70].

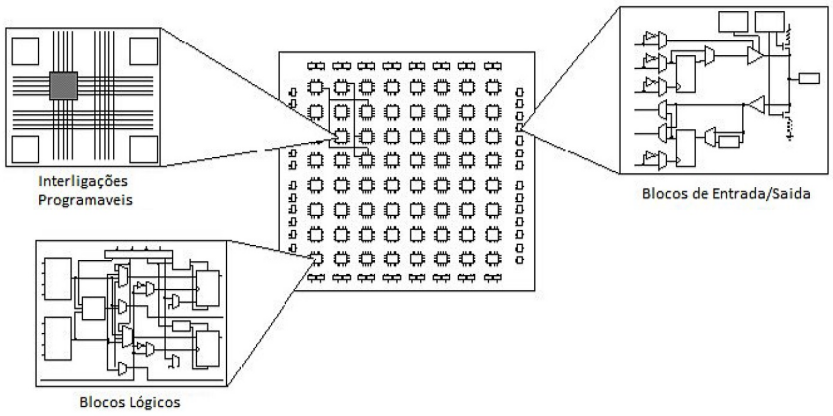


Figura 18 – Arquitetura interna de um FPGA. Fonte: (MEIXEDO, 2008)

Os blocos lógicos que compõem os FPGAs são geralmente baseados em arquitetura lookup table (LUT), que possibilita a implementação de qualquer função arbitrária com as entradas. Esses blocos são tipicamente agrupados em uma estrutura de matriz e interconectados de forma programável.

Cada bloco lógico contém normalmente uma LUT simples de três ou mais entradas e um flip-flop do tipo D, podendo variar para diferentes FPGAs. Um exemplo de bloco lógico em um FPGA pode ser visto na Fig. (19).

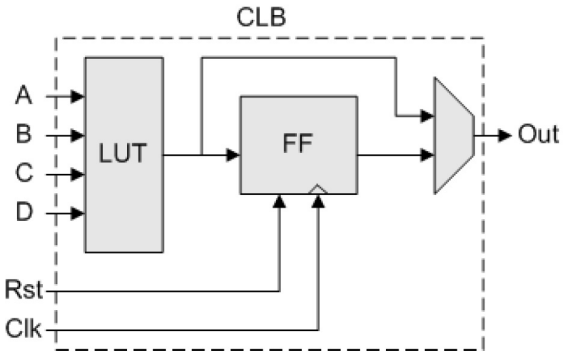


Figura 19 – Bloco lógico de um FPGA. Fonte: (MEIXEDO, 2008)

Funções mais complexas podem ser implementadas cascateando vários níveis de LUTs, conectando a saída de uma à entrada da próxima. A soma é um exemplo. Um somador completo iria necessitar duas LUTs de três entradas, ou 3-LUTs: uma para produzir a soma e outra para produzir o carry. Sendo uma operação tão comum, muitos fabricantes fornecem circuitos especializados dentro das células lógicas para realizar a soma completa, reduzindo os recursos necessários e facilitando a propagação do carry.

Multiplicação é também uma operação muito comum, principalmente em processamento digital de sinais (DSP). Um multiplicador pode ser implementado usando blocos lógicos para efetuar uma série de somas, de forma sequencial ou em paralelo. Sequencialmente, multiplicação demanda bastante tempo, necessitando de um ciclo de clock para cada bit do multiplicador realizar a soma associada. Realizar a multiplicação em paralelo melhora o timing, mas gasta significativamente mais recursos. Por isso, é comum em FPGAs com foco em DSP e outras aplicações de computação intensiva ter blocos de multiplicação e de acumulação dedicados, como os DSP48E (BAILEY, 2011).

Outra questão é o armazenamento, que é realizado usualmente nos flip-flops nas células lógicas. Enquanto isso é bom para dados que são acessados frequentemente, o espaço de armazenamento é limitado, o tornando caro para armazenar grandes volumes de dados. Muitos FPGAs modernos fornecem blocos de memória on-chip para buffers e armazenamentos intermediários Esses blocos são chamados de block RAM ou BRAM, muito utilizados por exemplo na construção de FIFOs (first-in first out) e de memórias cache personalizadas.

Também está se tornardo cada vez mais comum o design de arquiteturas mistas de *hardware* e software, com sistemas utilizando FPGAs juntamente com Microprocessadores. FPGAs modernos possibilitam a implementação de processadores soft-core, ou seja, implementados no FPGA, como o NIOS II em FPGAs da Altera, ou o Microblaze em FPGAs da Xilinx (BAILEY, 2011).

2.2.3 Linguagens de descrição de *hardware* e VHDL

A linguagem VHDL (Very high speed integrated circuit *Hardware* Description Language) foi originalmente desenvolvida como um método para documentar projetos de circuitos integrados de altíssima velocidade (VHSIC), em meados dos anos 1980.

Como esse projeto era de vital importância para o Departamento de Defesa Americano, foi feito um esforço de padronização por uma linguagem que pudesse descrever a estrutura e funcionalidade dos circuitos integrados, que fosse de fácil entendimento por qualquer projetista e possibilitasse simulações dos circuitos nela descritos.

Assim, optou-se por desenvolver uma linguagem que servisse como base para troca de informações sobre estes componentes e projetos. Linguagem essa que, independente do formato original do circuito, pudesse servir como uma descrição e documentação eficientes, padronizando a comunicação.

Além disso, projetos que utilizassem essa linguagem poderiam ser facilmente migrados de uma tecnologia para outra, de forma a possibilitar os avanços tecnológicos que ocorrem nessa área.

Esse esforço resultou na linguagem VHDL (VHSIC *hardware* description language) (LIPSETT; SCHAEFER; USSERY, 1989), que passou a ser um padrão aceito pelo IEEE.

No início da década de 1990, a VHDL foi usada primeiramente para projetos em ASIC e foram desenvolvidas ferramentas para automatizar o processo de criação e otimização das implementações. Na segunda metade da década, o uso de VHDL moveu-se para a área de dispositivos lógicos programáveis (CPLDs e FPGAs).

A linguagem VHDL possibilita descrições da interface e do comportamento, com declarações concorrentes e suporte a múltiplos níveis de hierarquia (descrição estrutural e comportamental).

Na descrição comportamental, o circuito é definido na forma de um algoritmo, utilizando construções similares às usadas por linguagens de programação convencional.

Quanto à descrição estrutural, tem-se uma visão mais próxima do *hardware*, com as seguintes características:

- Descreve apenas conexões entre componentes do projeto;
- Permite criar uma estrutura hierárquica de projeto;
- Descrição Top-down.

A sintaxe do VHDL foi herdada da linguagem ADA que, por sua vez, originou-se do PASCAL. O VHDL é essencialmente uma linguagem com descrições de operações exe-

cutadas em paralelo, exceto em blocos especiais onde se pode garantir o sequenciamento das instruções.

Basicamente, um programa escrito em VHDL é dividido em duas partes (descrição da interface – entity e do comportamento - architecture) mais a declaração das bibliotecas e pacotes utilizados, vejamos:

- Bibliotecas e pacotes: permitem agregar em um projeto definições de tipos de dados e funções previamente definidos;
- Entidades: uma entidade é qualquer componente VHDL que tenha um conjunto de portas de comunicação, com entradas e saídas. Uma entidade declara e/ou descreve as entradas e saídas do circuito (interface). É descrita por palavras reservadas, como por exemplo: ENTITY, PORT, IN, OUT, BUFFER, INOUT;
- Arquitetura: um conjunto de primitivas em VHDL que farão a efetiva descrição do *hardware*, ou seja, o modo de operação do circuito (relação entre interfaces). Define seu comportamento (ações) ou estrutura (conexões);
- Processos: um processo é basicamente o modelo de um componente, que possui uma lista de sinais dos quais depende (chamada lista de sensitividade). Os processos podem ser síncronos ou assíncronos e diversos, ou seja, dependentes ou não de um sinal de clock. Os processos descritos em uma arquitetura são sempre concorrentes, mas o fluxo dentro de um processo é sequencial. Utiliza as palavras reservadas, como: PROCESS, BEGIN e END PROCESS.

Outra característica importante da VHDL é o fato de ela ser uma linguagem fortemente tipada, permitindo a definição de novos tipos (TYPE IS). Exemplos de tipos pré-estabelecidos:

- BIT, BOOLEAN, CHARACTER
- INTEGER, REAL
- BIT_VECTOR, STRING
- RECORD
- STD_LOGIC**, STD_LOGIC_VECTOR**

Dentre as vantagens da utilização de uma linguagem de descrição de *hardware*, tem-se (CARRO, 2001):

- a descrição feita é bem definida, pois essas linguagens possuem regras sintáticas e semânticas que não permitem dupla interpretação;
- a descrição da especificação do circuito serve como documentação e explicita os objetivos do projeto;
- a padronização dessas linguagens resulta em portabilidade, tornando o código reutilizável em diferentes ambientes de desenvolvimento.

Por outro lado, alguns problemas permanecem ou são criados pela utilização dessas linguagens:

- investimento inicial em educação e treinamento dos projetistas;
- a síntese é limitada e muitos problemas devem ser particionados à mão.

2.2.4 FPGA e Processamento de imagens

FPGAs são inerentemente paralelos. Isso lhes provê a velocidade de um design de *hardware* enquanto mantêm a flexibilidade reprogramável do software, com um custo relativamente baixo. Assim, FPGAs são bem adequados para processamento de imagens, particularmente em baixo e médio nível, onde é possível explorar o paralelismo inerente em imagens digitais (BAILEY, 2011).

A maioria dos algoritmos de processamento de imagens consiste em uma sequência de operações. Tal estrutura sugere o uso de um módulo de processamento para cada operação, como mostrado na Fig. (20). Isso é chamado de arquitetura em pipeline, que é uma forma de paralelismo temporal. Essa arquitetura funciona de modo parecido a uma linha de produção, vez que os dados passam por cada estágio enquanto são processados. Cada módulo sucessor deve esperar até que o anterior complete seu processamento, embora o aumento de performance ocorra devido ao fato de que os módulos podem realizar operações em paralelo (SAMARAWICKRAMA, 2010).

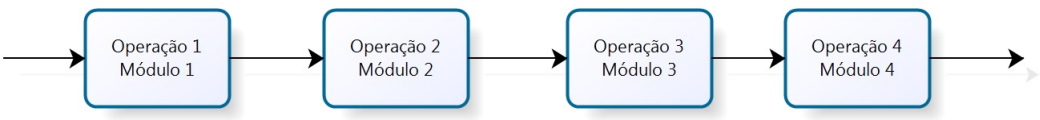


Figura 20 – Exemplo de arquitetura de pipeline.

Para uma arquitetura em pipeline, diferentes módulos de *hardware* são construídos para cada operação de processamento. Em um sistema síncrono, os dados passam simplesmente da saída de um módulo para a entrada de outro módulo.

Outra abordagem que pode ser usada em processamento de imagens é o paralelismo espacial, onde várias cópias do módulo de *hardware* são instanciadas para processar diferentes partes da imagem simultaneamente.

Um caso extremo de paralelismo espacial seria dedicar um módulo para cada pixel, o que não é muito prático, a não ser para imagens muito pequenas (BAILEY, 2011).

Paralelismo lógico dentro de operações em processamento de imagens, por sua vez, é também muito adequado a uma implementação em FPGA, acelerando significativamente o algoritmo. Isso é feito basicamente desenrolando laços, e realizando operações em paralelo quando possível. Um exemplo é a convolução, onde todas as multiplicações em uma janela ao redor de um pixel central podem ser efetuadas em paralelo em apenas um ciclo de clock.

2.2.5 Processamento multicamada

Como já mencionado, em processamento de imagens digitais, em geral, é quase sempre necessário agrupar uma certa quantidade de pixel ao redor de um pixel central (vizinhança) e realizar algumas operações, como comparações, multiplicações ou soma ponderada. É comum também que mais de um processamento desse tipo seja necessário, sendo que a entrada de um processo é a saída do anterior. Este é chamado de processamento multicamada, que é um dos maiores problemas em relação ao tempo de execução e consumo de memória de um sistema de visão computacional e processamento de imagens (Hsiao; Lu; Fu, 2010).

Em sistemas de processamento de imagens em tempo real, o dispositivo de aquisição de imagem envia normalmente os pixels serialmente, na chamada ordem raster-scan. Nessa ordem, os pixels são enviados linha por linha, da esquerda para a direita, como é mostrado na Fig (21). Desse modo, o processamento pode se realizado a medida que se recebe os pixels (processamento em stream).

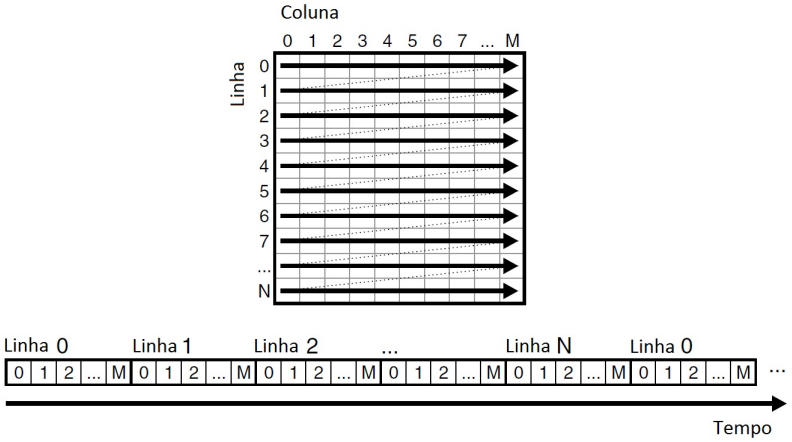


Figura 21 – Ordem raste-scan. Fonte: (BAILEY, 2011)

Uma das operações mais utilizadas em processamento de imagens, como já citado, é a convolução, onde o processamento é feito em vizinhança. Considerando que os pixels são recebidos em ordem raster-scan, percebe-se a necessidade de se acumular um determinado número de pixels para se começar o processamento. A Fig. (22) mostra a operação para o caso de uma máscara de tamanho igual a 5, onde faz-se necessário esperar quatro linhas de pixels e cinco pixels da quinta linha para começar a convolução.

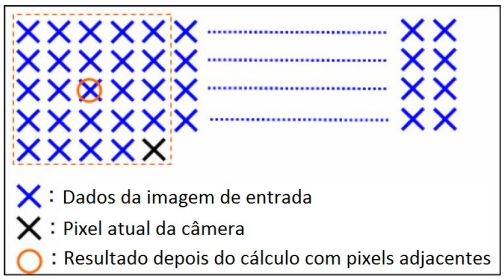


Figura 22 – Convolução com kernel 5 x5. Fonte: (Hsiao; Lu; Fu, 2010)

Um exemplo de processamento multicamada é a abertura de uma imagem ruidosa. O primeiro passo é filtrar o ruído. Logo após se realiza uma operação com um kernel de erosão. Por último, é feita uma operação com um kernel de dilatação. Analisando esse processamento, percebe-se que não há necessidade de esperar todos os resultados de um processo anterior para iniciar o próximo. Quando obtém-se uma certa quantidade de resultados da 1º camada, já é possível começar o processamento da 2ª camada, e o mesmo ocorre analogamente para a 3ª camada.

A Fig. (23) exemplifica essa afirmação. Nessa figura, há três processos para completar um processamento de imagem multicamada. O tamanho do kernel do 1º processo é 5x5, do 2º é 3x3, e do 3º é 3x3. Ao começar o processamento, os pixels da imagens irão chegar continuamente em ordem raster-scan, por meio de um buffer ou alguma outra interface com a câmera. Depois de acumular certa quantidade de pixels, começa-se a realização do 1º processo. Após a acumulação de certa quantidade de pixels do 1º processo, inicia-se paralelamente o 2º processo, e assim sucessivamente, formando um pipeline.

Assim, chega-se à conclusão que, quando o processamento de imagem da 1ª camada termina, o processo da 3ª camada é finalizado, após um certo tempo de latência. Isso se apresenta como uma vantagem para a implementação em *hardware*. Ainda, ao se fazer uma filtragem em um sistema baseado em CPU, por exemplo, não é possível realizar todas as operações de multiplicação em um ciclo de clock, o que é perfeitamente factível em um sistema de *hardware* (Hsiao; Lu; Fu, 2010).

O período que um processo qualquer deve esperar um processo anterior pode ser definido pela Eq. (2.11), onde O_p é a imagem resultante do processo p, K_p é a dimensão

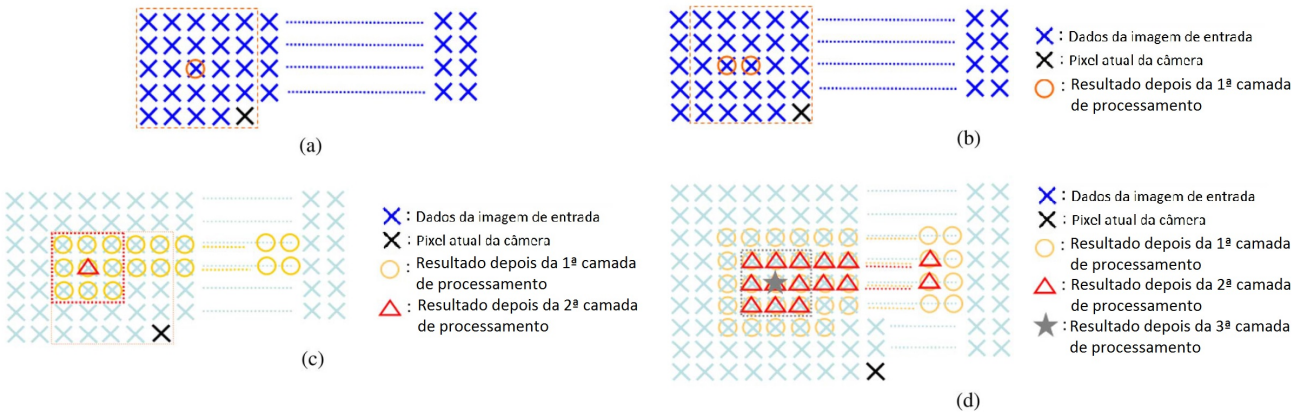


Figura 23 – Exemplo de processamento multicamada. Fonte: (Hsiao; Lu; Fu, 2010)

do kernel do processo p (por exempo 3 x 3 ou 5x5) e I_{wp} é a largura de O_p.

$$T_w(p) = (K_p - 1) * Iw_p + K_p, \tag{2.11}$$

$T_w(p)$ é o tempo de espera do processo p, e I_{wp} pode ser definido pela Eq. (2.12), sendo I_{w0} a largura da imagem inicial.

$$Iw_p = \begin{cases} Iw_0 - K_p + 1, p = 1 \\ Iw_{p-1} - K_p + 1, p > 1 \end{cases} \tag{2.12}$$

Observa-se que, nesse caso, a imagem resultante será menor que a imagem inicial, ou seja, a borda será desprezada, o que não se configura como um problema para a maioria dos algoritmos.

3 Descrição da implementação

Nesse capítulo, será descrita a implementação do algoritmo de detecção de cantos de Harris. A arquitetura de *hardware* desenvolvida em FPGA será apresentada detalhadamente.

A estrutura do presente capítulo está formatada de maneira a se abordar, na ordem em que se seguem, os seguintes itens: como se procede o processo de desenvolvimento para a implementação, as plataformas de desenvolvimento e o kit FPGA utilizado, a implementação do detector de cantos de Harris em *software* Matlab e, por fim, a arquitetura de *hardware* desenvolvida, na qual primeiramente será abordado o sistema como um todo e, posteriormente, cada bloco em detalhes.

3.1 Processo de Desenvolvimento

De acordo com (SASS; SCHMIDT, 2010), um algoritmo de processamento de imagens não deve ser desenvolvido diretamente no FPGA, pois o ciclo de desenvolvimento é muito grande para permitir um desenvolvimento interativo. Linguagens de descrição de *hardware*, como VHDL, são de propósito geral e permitem um controle considerável sobre o circuito a ser implementado. A vantagem disso é que o desenvolvedor consegue utilizar recursos específicos do FPGA, otimizar o *design* para velocidade ou para quantidade de recursos utilizados de acordo com o estilo de programação. Assim, é possível implementar designs rápidos e eficientes.

Uma desvantagem é que o desenvolvedor deve controlar tudo nos mínimos detalhes, tanto o caminho de dados quanto a lógica de controle. Dessa forma, programar algoritmos complexos se torna uma tarefa complicada sem o devido planejamento. Por isso, muitos algoritmos são primeiramente desenvolvidos em linguagem de *software* de mais alto nível, como C ou MATLAB. Uma das principais vantagens de se separar o desenvolvimento do algoritmo da implementação em FPGA é que a aplicação pode ser rigorosamente testada e validada antes da tarefa mais complexa de mapear o algoritmo no *hardware* de destino (BAILEY, 2011). Esse tipo de validação é geralmente muito mais fácil de ser realizado em um ambiente baseado em *software*. Os testes em *hardware* resumem-se, então, em verificar se as operações foram executadas corretamente. Se todas as operações funcionam satisfatoriamente, significa que todo o algoritmo de processamento de imagens, na maioria dos casos, também funcionará.

O processo de desenvolvimento deste trabalho foi baseado nessas ideias. A Fig. (24) mostra o fluxo de projeto utilizado para a implementação.

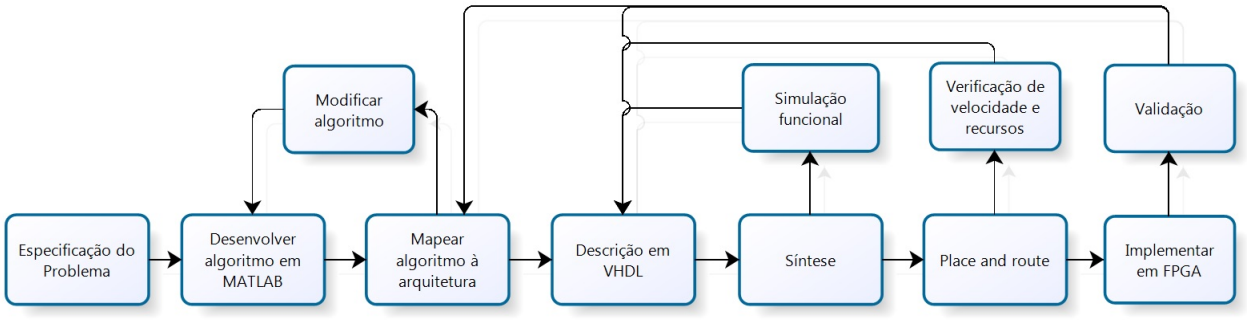


Figura 24 – Fluxo de projeto utilizado.

Seguem abaixo esclarecimentos de cada etapa do projeto:

- Especificação do problema: realiza-se um estudo detalhado do algoritmo de detecção de cantos de Harris e de suas etapas de execução;
- Desenvolvimento do algoritmo em MATLAB: o algoritmo é desenvolvido em ambiente MATLAB, que permite uma maior abstração e uma primeira validação do algoritmo;
- Mapeamento do algoritmo à arquitetura: analisa-se as possíveis implementações em *hardware*, a fim de aproveitar os benefícios de paralelismo;
- Modificação do algoritmo: caso a arquitetura projetada não esteja satisfatória, retorna-se ao desenvolvimento do algoritmo em ambiente MATLAB;
- Desenvolvimento do *design* em VHDL: implementa-se o projeto de arquitetura em linguagem de descrição de *hardware* VHDL;
- Síntese: realiza-se a síntese do código VHDL, por meio do sintetizador XST, da Xinlix;
- Simulação funcional: realiza-se a validação em nível de síntese, juntamente com o progresso da implementação;
- Place and route: faz-se a alocação do *hardware* no FPGA;
- Verificação de velocidade e recursos: verifica-se se o *hardware* descrito foi sintetizado como desejado;
- Implementação em FPGA: configura-se o FPGA com o bitstream;
- Validação: valida-se a implementação em FPGA com o auxílio da ferramenta ChipScope, a qual insere um analisador lógico no *design*.

3.2 Kit e plataformas de desenvolvimento

O Matlab foi escolhido como ferramenta para uma primeira implementação do algoritmo. Esse *software* é conhecido mundialmente como uma excelente ferramenta para soluções de problemas matemáticos, científicos e tecnológicos, e possui comandos muito próximos da forma como se escrevem as expressões matemáticas. A linguagem é ideal para desenvolver rapidamente protótipos de novos programas.

O Matlab começou apenas como um *software* para operações matemáticas sobre matrizes, mas ao longo dos anos transformou-se em um sistema computacional flexível capaz de desenvolver essencialmente qualquer problema técnico. MATLAB possui uma vasta biblioteca de funções predefinidas, tais como: matemática elementar; funções especiais; matrizes elementares; matrizes especiais; decomposição e fatorização de matrizes; análise de dados; polinômios; solução de equações diferenciais; equações não-lineares e otimização; integração numérica; processamento de sinais entre outras.

Além disso, o Matlab disponibiliza o pacote Image Processing Toolbox, o qual caracteriza-se por uma coleção de funções que estende a capacidade de desenvolvimento para processamento de imagens. Essas funções tornam operações de processamento de imagens simples de se descrever, proporcionando, assim, um ambiente ideal para protótipo rápido de *software* para soluções em processamento de imagens (Gonzalez; Woods; Eddins, 2009).

Para o desenvolvimento em FPGA, utilizou-se o kit de desenvolvimento da Xilinx XUPV5-LX110T (Fig. 25). A fabricante Xilinx foi uma das primeiras desenvolvedoras da tecnologia FPGA, possuindo várias famílias de dispositivos, onde se destaca a série Spartan e Virtex. A grande diferença entre as duas famílias é que os dispositivos Spartan são desenvolvidos com foco em baixo custo, enquanto a série Virtex é projetada com foco em alta performance.

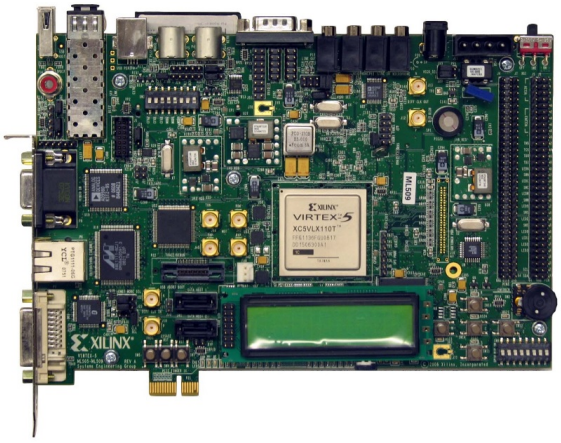


Figura 25 – Kit de desenvolvimento XUPV5-LX110T.

O kit XUPV5-LX110T é uma plataforma unificada para ensino e pesquisa, em

áreas como: sistemas embarcados, processamento digital de sinais, sistemas operacionais, aplicações de rede, processamento de imagens e vídeo, dentre outros.

O mesmo kit possui diversas interfaces, como Ethernet, RS232, VGA e USB, podendo ser usado para uma grande variedade de aplicações. Possui, ainda, o FPGA XC5VLX110T, da linha Virtex 5.

A família Virtex 5 apresenta alto desempenho, utilizando arquitetura LUT de seis entradas, ou 6-LUT, tendo até 36-Kbit de block RAM/FIFOs disponíveis e 64 módulos DSP48E de alta performance.

Para a implementação com o kit, utilizaram-se as ferramentas de desenvolvimento e validação da Xilinx. O *software* escolhido prioritariamente como base para o desenvolvimento foi o ISE Project navigator, onde descreveu-se todo o código VHDL produzido e fez-se o uso do sintetizador XST da Xilinx. Ainda, usaram-se ferramentas de simulação e validação da Xilinx, como ISIM e o ChipScope.

3.3 Algoritmo de Harris em Matlab

Nesta seção, a implementação realizada do algoritmo de detecção de cantos de Harris em MATLAB será descrita.

O Detector de Cantos de Harris pode ser resumido nos seguintes passos:

- 1º - Realizar a leitura de uma imagem;
- 2º - Calcular as derivadas nas direções x e y, utilizando as máscaras de Sobel (Eq. 3.1);

$$I_x = S_x * I \quad I_y = S_y * I \tag{3.1}$$

- 3º - Computar os produtos das derivadas (Eq. 3.2);

$$I_{x2} = I_x.I_x \quad I_{y2} = I_y.I_y \quad I_{xy} = I_x.I_y \tag{3.2}$$

- 4º - Realizar a convolução dos produtos das derivadas com uma máscara gaussiana (Eq. 3.3):

$$G_{x2} = G * I_{x2} \quad G_{y2} = G * I_{y2} \quad G_{xy} = G * I_{xy} \tag{3.3}$$

- 5º - Para cada pixel *(x,y)*, calcular a resposta do detector (Eq. 3.4);

$$G(x,y) = \begin{bmatrix} G_{x2}(x,y) & G_{xy}(x,y) \\ G_{xy}(x,y) & G_{y2}(x,y) \end{bmatrix}$$
$$R = (G_{x2}G_{y2} - G_{xy}^2) - k(G_{x2} + G_{y2})^2 \tag{3.4}$$

- 6º - Realizar a limiarização sobre o valor de R.

O MATLAB possui várias funções já predefinidas que possibilitam a implementação desse algoritmo de forma bem direta. Inicialmente, faz-se a leitura de uma imagem armazenada em disco por meio da função imread, onde um arquivo de imagem é carregado no ambiente do MATLAB em forma de matriz. Logo após, definem-se as máscaras de Sobel para as derivadas nas direções horizontal e vertical, de tamanho 3x3, como podem ser vistas na Eq. (3.5).

$$S_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad S_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{3.5}$$

Realiza-se, então, a filtragem da imagem inicial com as máscaras de Sobel, por meio da operação de convolução. A convolução é realizada de modo direto em MATLAB, por meio da função conv2. Essa função recebe como parâmetro a imagem inicial e a máscara para a convolução. O tamanho da imagem resultante é determinado por um dos parâmetros. No caso, escolheu-se a opção 'valid', onde as bordas da imagem de saída foram desconsideradas, gerando uma imagem ligeiramente menor, como já explicado anteriormente. Essa opção foi definida por não ser muito interessante, na maioria das aplicações, que o algoritmo encontre cantos nas bordas das imagens.

Obtém-se, desse modo, as derivadas da imagem nas direções horizontal e vertical, respectivamente Ix e Iy. O próximo passo é calcular o produto das derivadas. Calcula-se então Ix2, Iy2 e Ixy, multiplicando as derivadas. Depois disso, define-se a máscara de gauss, de tamanho 5x5. Escolheu-se aqui uma máscara pseudo-gaussiana, mostrada na Eq. (3.6), que possui somente potências de 2. Esse fato ocorreu visando uma implementação em *hardware* mais eficiente, onde operações de multiplicação podem ser substituídas por operações de deslocamento.

$$Gauss = \begin{bmatrix} 1 & 2 & 4 & 2 & 1 \\ 2 & 4 & 8 & 4 & 2 \\ 4 & 8 & 16 & 8 & 4 \\ 2 & 4 & 8 & 4 & 2 \\ 1 & 2 & 4 & 2 & 1 \end{bmatrix} \tag{3.6}$$

Realiza-se, dessa forma, a convolução dos produtos das derivadas com essa máscara de Gauss, tendo como resultado as imagens Gx2, Gy2 e Gxy, que são as convoluções com Ix2, Iy2 e Ixy, respectivamente. Também, para essas convoluções, são desprezadas as bordas utilizando-se a opção 'valid' na função conv2. Faz-se necessário calcular o fator de Harris, que é definido pela Eq. (3.4). O fator é dependente dos componentes Gx2, Gy2 e Gxy e do fator k. Essa equação é descrita de maneira direta em MATLAB. O resultado é uma imagem com o fator de Harris para cada coordenada. Esse é o resultado que se espera obter com a implementação em FPGA. Por fim, é feita uma limiarização nos resultados do fator de Harris, que pode variar de acordo com a aplicação.

O código em MATLAB se encontra, em anexo, para análise.

3.4 Algoritmo de Harris em *Hardware*

3.4.1 Sistema completo

Tendo o algoritmo sido implementado em Matlab, o passo seguinte foi a realização do *design* de uma arquitetura de *hardware* capaz de implementar o algoritmo de forma eficiente, utilizando recursos de paralelismo e pipelining, quando possível.

Como já mencionado, a arquitetura de *hardware* foi descrita utilizando-se a linguagem VHDL. A descrição foi toda parametrizada, a fim de se implementar a arquitetura em vários tamanhos de imagens diferentes e de se ter a possibilidade de mudar parâmetros internos do algoritmo, como os tamanhos do kernel e seus valores, quantidade de bits de registradores e de portas de comunicação entre blocos. Esses parâmetros foram declarados em um pacote VHDL (package) como constantes. A vantagem dessa abordagem é que, caso se deseje realizar alguma mudança, não será necessário ter de realizá-la no código, o que geraria uma propensão a erro.

Ao analisar o algoritmo de detecção de cantos de Harris, observa-se que é possível fazer várias otimizações para usufruir do paralelismo em FPGA. As operações que não são dependentes diretamente podem ser feitas em paralelo. Um exemplo disso é a multiplicação de uma máscara em uma vizinhança, que pode ser feita em paralelo de uma só vez. A Fig. (26) mostra a arquitetura proposta.

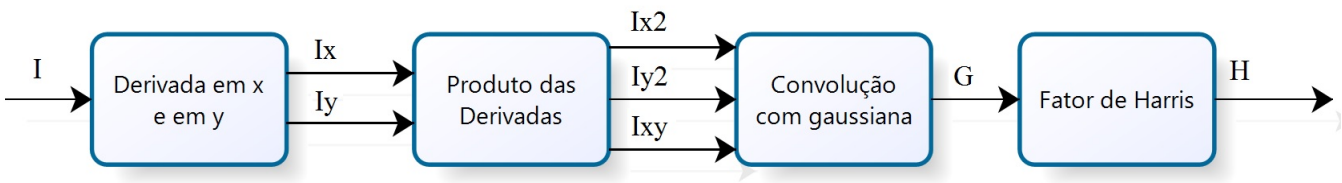


Figura 26 – Detector de Harris desenvolvido em FPGA.

A imagem de entrada é recebida sequencialmente em ordem raster-scan, numa taxa de até um pixel por clock. O processamento é feito totalmente em tempo real, em um processamento multicamada ao longo de um pipeline. Os resultados intermediários das operações são armazenados em registradores, evitando assim o uso de memória externa e garantindo uma maior frequência de clock no *design*.

Na primeira camada, calculam-se as derivadas de primeira ordem da imagem, nas direções horizontal e vertical, em paralelo, utilizando-se os kernels de Sobel de tamanho 3 x 3, já apresentados na última seção.

Tendo as derivadas Ix e Iy, calculam-se, na segunda camada, os produtos das derivadas, também de forma paralela.

Na terceira camada, realizam-se as três convoluções dos produtos das derivadas com o kernel de gauss em paralelo, obtendo-se a matriz de autocorrelação G (Gx2, Gy2 e Gxy).

Na quarta e última camada, calcula-se o fator de Harris.

O algoritmo foi dividido em módulos, de forma a facilitar a implementação e a validação, bem como proporcionar portabilidade e reusabilidade para futuros projetos. Pensando nisso, a interface entre os módulos foi realizada utilizando-se um simples protocolo, com objetivo de padronizar a comunicação. A Fig. (27) mostra o exemplo de um módulo genérico.

Figura 27 – Módulo genérico utilizado.

O sinal mais importante é o sinal *data*, cujo tamanho varia dependendo do módulo. Esse sinal, na entrada ou na saída, é válido quando os sinais *data_in_valid* ou *data_out_valid*, respectivamente, são ativados. A seguir, cada interface será explicada:

- *data_in*: vetor de bits contendo os dados de entrada do módulo;
- *data_in_valid*: valor lógico para representar uma entrada válida. '1' quando *data_in* está disponível;
- *data_out*: vetor de bits contendo os dados de saída do módulo;
- *data_out_valid*: valor lógico para representar uma saída válida. '1' quando *data_out* está disponível;

A Fig. (28) mostra um exemplo do protocolo.

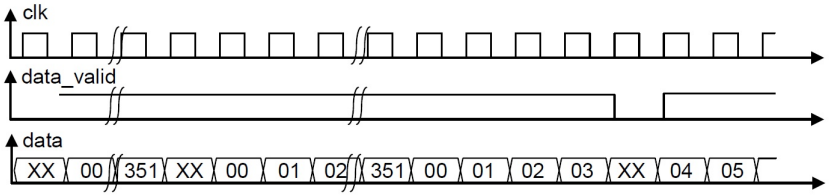


Figura 28 – Exemplo do protocolo utilizado entre módulos.

Nas seções que se seguem, a convolução em *hardware* será abordada, e cada módulo será explorado com mais propriedade.

3.4.2 Convolução em *Hardware*

O algoritmo de detecção de Harris, já detalhado, faz bastante uso da operação de convolução. Embora a convolução seja facilmente implementada em linguagens de *software* de alto nível, como foi o caso da implementação em MATLAB, o mesmo não acontece em descrição de *hardware*. Foi dada, então, uma atenção especial a esse bloco.

A convolução 2D é um dos blocos fundamentais do detector de cantos de Harris, como também de diversas aplicações em processamento de imagens e de vídeo. Sua implementação requer uma alta demanda computacional e o uso intensivo de memória em sistema tradicional baseado em CPU, como discutido anteriormente.

Tornou-se, por isso, imprescindível a realização de um projeto de uma arquitetura de *hardware* eficiente para sua implementação. Na convolução, faz-se necessário que se obtenham as vizinhanças das imagens para a realização dos cálculos. Como a imagem é recebida pixel a pixel, em ordem raster-scan, não é possível se realizar a convolução diretamente, em contraste ao que foi feito na implementação em MATLAB, onde a imagem já estava toda disponível em memória. Como explicado anteriormente, precisa-se se obter uma certa quantidade de pixels antes de começar a convolução em si. O *hardware* que realiza essa tarefa foi denominado arquitetura de vizinhança.

Obtida a vizinhança, faz-se necessária a realização da multiplicação da vizinhança com a máscara e a soma de todos os valores resultantes. Desenvolveu-se uma arquitetura para efetuar essas duas operações, chamada de produto interno. A Fig. (29) mostra o módulo de convolução.

A convolução desenvolvida baseia-se, dessa forma, em dois blocos: a arquitetura de vizinhança e o produto interno, os quais serão detalhados a seguir.

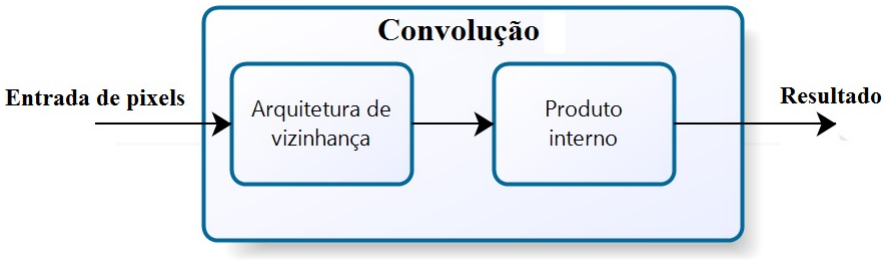


Figura 29 – Bloco de convolução bidimensional.

3.4.2.1 Arquitetura de vizinhança

A arquitetura de vizinhança é um componente essencial para a realização da convolução 2D em *hardware* em tempo real.

Considerando um filtro com janela 3 x 3, para se iniciar a convolução, precisa-se ter a primeira vizinhança de nove pixels. Para isso, faz-se necessário que se receba as duas primeiras linhas da imagem e mais três pixels da terceira linha, para se obter a primeira vizinhança. De modo análogo, para um filtro de tamanho 5 x 5, seria primordial obter quatro linhas completas da imagem e mais cinco pixels da quinta linha, para se ter a primeira vizinhança. Para se realizar esse processo em *hardware*, é essencial utilizar estruturas para o armazenamento temporário de linhas da imagem e de registradores de deslocamento. A arquitetura de vizinhança implementada é mostrada na Fig. (30), para uma vizinhança 3 x 3.

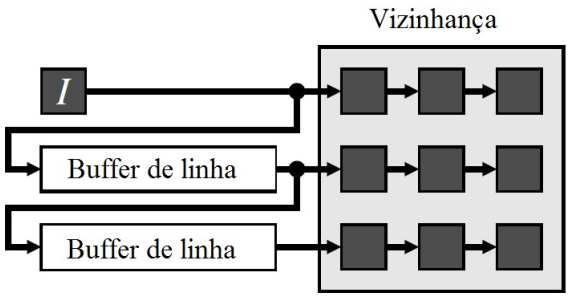


Figura 30 – Arquitetura de vizinhança. Fonte: (BAILEY, 2011)

A arquitetura de vizinhança é implementada utilizando buffers de linha e registradores de deslocamento. Os buffers de linha têm por objetivo realizar o atraso dos pixels disponíveis constantemente.

Essa arquitetura funciona da seguinte maneira: os pixels entram constantemente no primeiro buffer; ao se completar uma linha e o buffer se preencher, o segundo buffer será acionado e os dados do primeiro buffer começarão a ser transferidos ao segundo, e assim for diante, dependendo do número de buffers, que é diretamente dependente do tamanho do kernel. Assim que o último buffer estiver completo, os pixels começarão a passar para

os registradores de deslocamento, obtendo-se, desse, modo, a vizinhança. Após obtida a primeira vizinhança, a cada novo pixel de entrada, uma nova vizinhança estará disponível. Os buffers de linha são implementados como arquiteturas do tipo FIFO (First In First Out) síncronas utilizando block RAMs do FPGA.

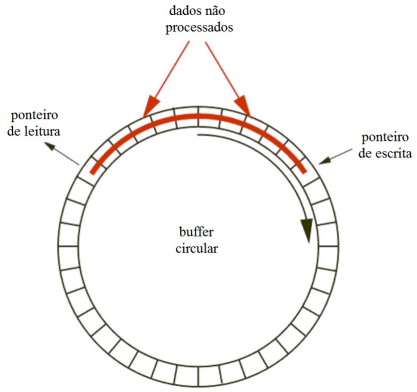


Figura 31 – Exemplo de operação de um buffer circular.

Uma arquitetura FIFO é descrita em *hardware* por meio de um buffer circular, que é um espaço de memória dedicado, com dois ponteiros, um de escrita e um de leitura. Quando é feita uma requisição de escrita, o pixel é escrito na posição apontada pelo ponteiro de escrita e esse ponteiro é acrescido de uma posição. Quando é requisitada uma operação de leitura, o ponteiro de leitura é acrescido também de uma posição. Assim, quando os ponteiros estiverem na mesma localização, a FIFO estará vazia. Quando o ponteiro de escrita acrescido de uma posição for igual ao ponteiro de leitura, a FIFO estará cheia. A Fig. (31) mostra um exemplo de um buffer circular.

Uma unidade de controle foi desenvolvida para controlar os sinais de entrada e habilitar o sinal de válido quando a vizinhança estiver completa. Essa unidade de controle recebe o sinal de *data_in_valid* e possui um contador, que, dependendo da contagem, ativa o sinal *data_valid_out*.

3.4.2.2 Produto interno

No módulo de produto interno, tendo dois vetores $\mathbf{a} = \begin{bmatrix} a_1 & a_2 & \cdots & a_n \end{bmatrix}$ e $\mathbf{b} = \begin{bmatrix} b_1 & b_2 & \cdots & b_n \end{bmatrix}$ se realiza a operação definida pela Eq. (3.7).

$$\mathbf{a}.\mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots a_n b_n \tag{3.7}$$

As entradas desse módulo são o kernel de convolução e a vizinhança obtida com a arquitetura descrita anteriormente.

A primeira tarefa é efetuar a multiplicação de cada elemento da vizinhança com a máscara. O kernel deve ser rebatido antes da multiplicação. Observando a Eq. (3.7),

percebe-se a possibilidade de se realizar todas as multiplicações da vizinhança com o kernel em paralelo. Além disso, se os valores do kernel forem potência de 2, não faz-se necessário computar a multiplicação. É possível realizar apenas operações de deslocamento aritmético, o que economiza os multiplicadores dedicados do FPGA. Para tornar a descrição VHDL genérica para qualquer kernel, a operação descrita foi de multiplicação, sendo que geralmente o sintetizador interpreta a multiplicação como um deslocamento quando o operador é uma potência de dois.

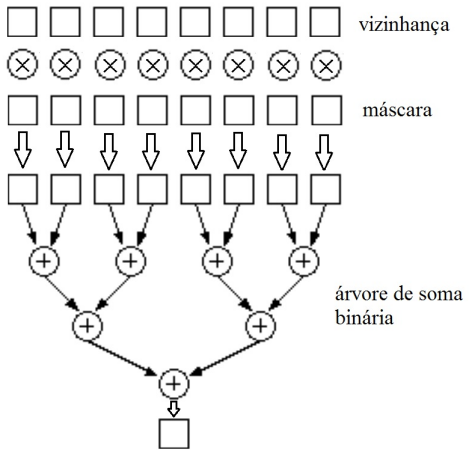


Figura 32 – Arquitetura do módulo de produto interno.

Depois de executar a multiplicação em paralelo, os resultados devem ser somados. Para aumentar o desempenho, desenvolveu-se uma árvore binária de soma. As somas são realizadas aos pares e em um pipeline. Isso garante um menor atraso no caminho combinacional e, consequentemente, uma maior frequência de clock do circuito. O número de estágios do pipeline é de log2(N), onde N é o número de entradas da árvore de soma, que no caso da implementação é o número de elementos do kernel. Para um kernel 3 x 3, por exemplo, tem-se um N igual a 9 e 4 estágios de pipeline. Para uma vizinhança de tamanho 5 x 5, N é igual a 25. A Fig. (32) mostra a arquitetura do *hardware* de produto interno.

3.4.3 Derivadas da imagem

O primeiro subsistema do Detector de Cantos de Harris calcula as derivadas da imagem nas direções horizontal e vertical usando a convolução 2D. A Fig. (33) mostra esse módulo.

Utiliza-se, primeiramente, uma arquitetura de vizinhança 3x3 para receber o stream de pixels da imagem inicial. Não é preciso haver duas arquiteturas de vizinhança, pois uma mesma vizinhança pode ser utilizada para calcular as derivadas em x e em y.

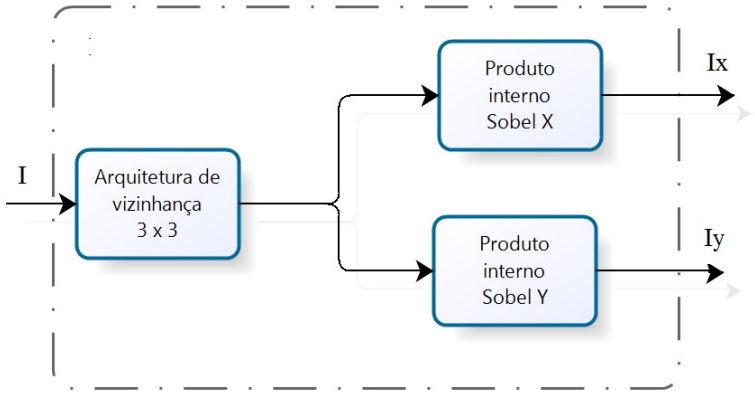


Figura 33 – Arquitetura para calcular derivadas x e em y da imagem de entrada.

Quando a vizinhança estiver válida, ela será repassada para os blocos de produto interno, para a realização do cálculo paralelo de Ix e Iy, onde serão utilizadas as máscaras de Sobel, anteriormente mostradas na Eq. (3.5).

3.4.4 Produtos das derivadas e convolução com janela gaussiana

Os produtos das derivadas são calculados em paralelo, por meio de três multiplicadores, onde se obtém Ix2, Iy2 e Ixy.

A próxima etapa é realizar a convolução de cada produto com o kernel de Gauss (Eq. (3.6)), para se obter a matriz de autocorrelação.

A primeira ideia para a realização dessas convoluções foi instanciar uma arquitetura de vizinhança para cada produto separadamente. Percebeu-se que isso não era preciso. Era possível instanciar apenas uma arquitetura de vizinhança, com FIFOs e registradores de maior tamanho de dados, para acomodar os três produtos. Isso foi feito então para uma vizinhança de 5x5, que foi o tamanho de kernel de gauss utilizado. Os produtos Ix2, Iy2 e Ixy foram, dessa maneira, concatenados e colocados como entrada da arquitetura de vizinhança.

Quando a vizinhança está disponível, faz-se necessário realizar a separação de cada componente dos produtos para a convolução com o kernel de gauss. Na sequência, são instanciadas três arquiteturas de produto interno que atuam em paralelo. Quando os resultados estão disponíveis, Gx2, Gy2 e Gxy são passados ao módulo de Harris. A Fig. (34) mostra esse módulo.

3.4.5 Fator de Harris

Depois de calcular a matriz de autocorrelação, é essencial determinar o fator de Harris, como descrito anteriormente. A Eq. (3.8), repetida aqui, deve ser calculada em

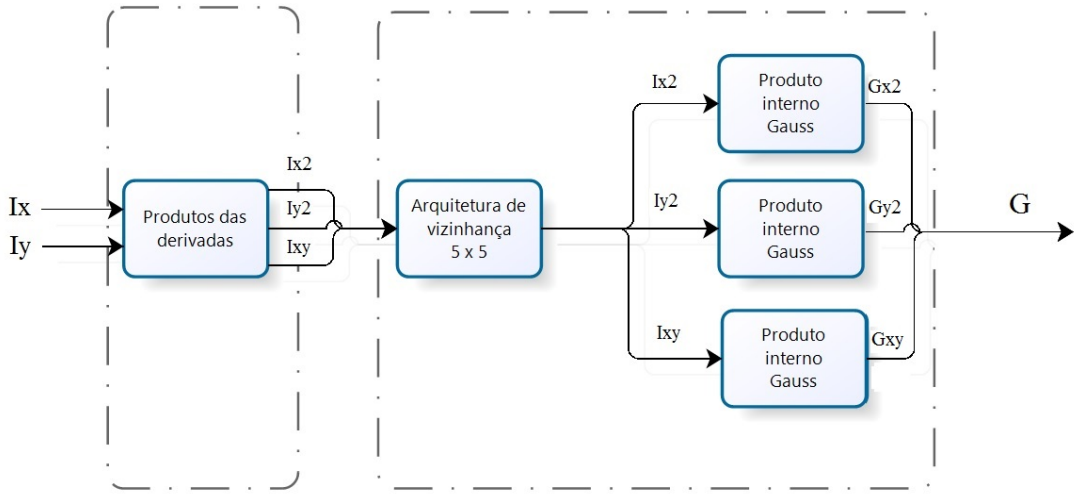


Figura 34 – Arquitetura para o cálculo dos produtos das derivadas e convoluções com janela gaussiana.

hardware. Para que o atraso combinacional fosse o menor possível, a implementação foi totalmente feita em um pipeline, de forma a quebrar o caminho crítico.

$$R = (G_{x2}G_{y2} - G_{xy}^2) - k(G_{x2} + G_{y2})^2 \tag{3.8}$$

Utilizou-se um pipeline de três estágios. As operações de soma, subtração e multiplicação foram divididas e realizadas em paralelo, quando possível. Como já mencionado, é recomendado que k tenha um valor entre 0.03 e 0.08, sendo 0.04 o valor que retorna melhores resultados, de acordo com a literatura. O fator k de multiplicação na equação foi aproximado por uma operação de deslocamento. Realizou-se uma operação de deslocamento à direita (right shift) de cinco bits, que é similar a uma multiplicação por aproximadamente 0.0321. Isso evitou a necessidade de se trabalhar com ponto fixo.

4 Resultados

4.1 Validação

A validação foi realizada de forma conjunta ao processo de desenvolvimento. Isso facilitou bastante a localização e a correção de erros.

Figura 35 – Detecção de cantos em MATLAB para imagem de tamanho 512 x 512.

A Fig. (35) mostra a detecção de cantos realizada em ambiente MATLAB, para a imagem lena, de tamanho 512 x 512, muito conhecida na área de processamento digital de imagens. No quadrante superior esquerdo, apresenta-se a imagem original. No quadrante superior direito, pode-se ver a imagem resultante, após o cálculo do fator de Harris, onde áreas mais claras correspondem a regiões de melhores respostas aos cantos. A próxima etapa é realizar uma operação de limiarização sobre essa imagem. Também, valores não máximos numa vizinhança são desconsiderados, podendo ser observado esse resultado na imagem localizada no quadrante inferior esquerdo da Fig. (35). Isso evita a existência de muitos cantos próximos uns aos outros. Por fim, no quadrante inferior direito da Fig. (35), mostra-se a imagem original juntamente com os cantos, em vermelho.

Para realizar testes em *hardware*, foi necessária a implementação de uma interface capaz de enviar pixels de imagens para o *hardware* desenvolvido, de forma a simular uma

câmera. Desenvolveu-se, então, um módulo de *hardware*, chamado send_pixels, com o objetivo de armazenar uma imagem digital e enviar seus pixels em ordem raster-scan. A imagem digital foi armazenada em uma memória ROM (somente de leitura), instanciada dentro do FPGA. Realizou-se, então, a interface do módulo criado com o *hardware* de detecção de cantos. Adicionalmente, desenvolveu-se uma lógica de controle para enviar os pixels dessa imagem sequencialmente, numa taxa de até um pixel por ciclo de clock, para o Detector de Cantos de Harris, em ordem raster-scan. A Fig. (36) mostra o sistema aplicado.

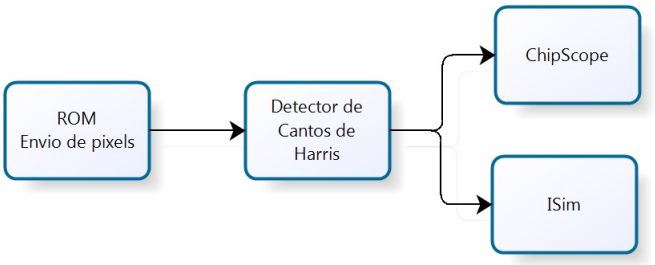


Figura 36 – Sistema utilizado para validação.

Os testes foram aplicados em dois níveis. Primeiramente, em nível de síntese, onde realizaram-se simulações com o auxílio do *software* ISim, da Xilinx. Nesse simulador, foi possível selecionar os sinais que se desejava observar, aplicar estímulos ao *hardware* e analisar a saída, bem como selecionar o período de clock que o sistema devia operar.

Em um estágio mais avançado de desenvolvimento, a validação foi feita em FPGA, utilizando-se a ferramenta ChipScope, quando foi possível se verificar de forma mais robusta o funcionamento do detector. Essa ferramenta foi bastante útil durante o processo de desenvolvimento, pois inseriu um analisador lógico dentro do *design*, permitindo verificar certos sinais do *hardware* implementado em tempo real.

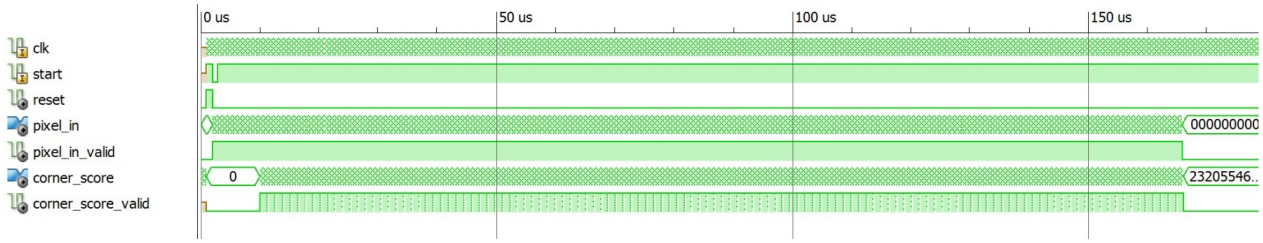


Figura 37 – Simulação de processamento de imagem de tamanho 128 x 128.

O ChipScope possibilitou exportar os dados obtidos em testes para um arquivo em formato ASCII. Assim foi realizado. Escreveu-se, então, um script em MATLAB para se ler o arquivo resultante desse processo. Isso possibilitou a reconstrução da imagem resultante do processamento em *hardware*, como também a comparação com imagens

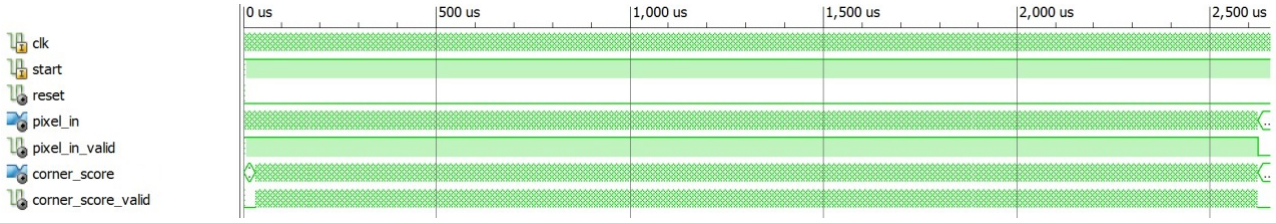


Figura 38 – Simulação de processamento de imagem de tamanho 512 x 512

obtidas do processamento MATLAB. Desse modo, a verificação foi realizada, de forma a confirmar o correto funcionamento da implementação em FPGA.

As Figs. (37) e (38) mostram a simulação no *software* ISIM do processamento de imagens de tamanhos 128 x 128 e 512 x 512, respectivamente. Os pixels são enviados sequencialmente pelo módulo criado. Após um curto período, inicia-se o recebimento dos resultados referentes ao fator de Harris, de modo paralelo. Assim, ao mesmo tempo em que se recebe os pixels, as respostas ao fator de Harris são disponibilizados.

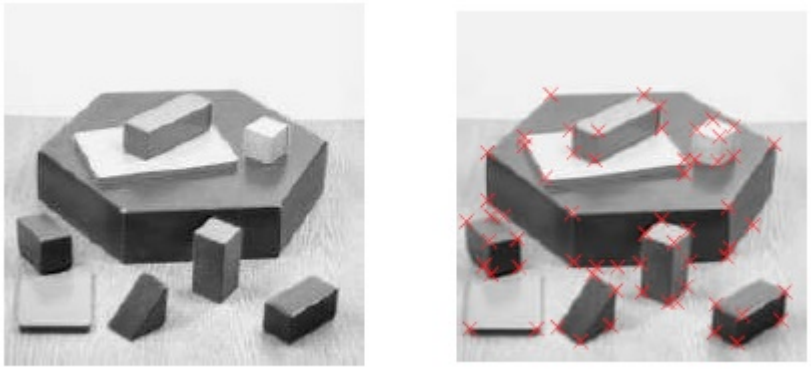


Figura 39 – Detecção de cantos em *hardware* realizada em imagem contendo diversos blocos.

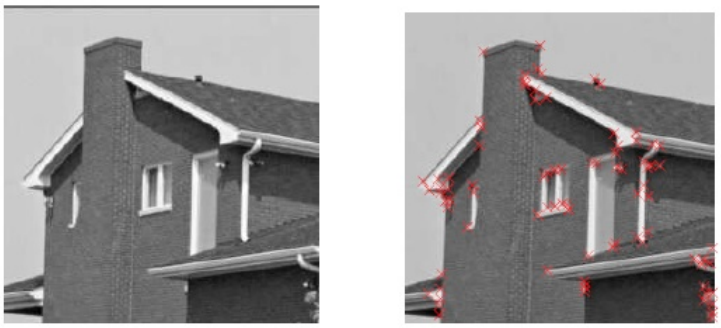


Figura 40 – Detecção de cantos em *hardware* realizada em imagem de uma casa.

Quando se finaliza o processo de recebimento de pixels, o processamento chega ao seu fim, logo após um curto espaço de tempo. Percebe-se, desse modo, que quanto maior

for o tamanho da imagem, mais díficil se torna notar a latência do processamento, devido ao aumento significativo de dados.

As Figs. (39) e (40) mostram duas imagens de teste, onde realizou-se o processamento diretamente em *hardware*. O correto funcionamento da implementação em FPGA foi verificado, tendo sido obtidas imagens idênticas àquelas em *software*.

4.2 Desempenho

Com a implementação realizada no FPGA do tipo XUPV5-LX110T, obteve-se uma frequência máxima no *design* de aproximadamente 100MHz, como pode ser visto na Fig. (41), obtida com o sintetizador XST. Observou-se, porém, que à medida em que foram feitas modificações na descrição do *hardware*, houve uma variação dessa frequência entre aproximadamente 90MHz e 130Mhz.

```
Timing Summary:
---------------
Speed Grade: -1

   Minimum period: 9.550ns (Maximum Frequency: 104.712MHz)
   Minimum input arrival time before clock: 3.591ns
   Maximum output required time after clock: 3.259ns
   Maximum combinational path delay: No path found

=========================================================================
```

Figura 41 – Frequência máxima de clock média suportada por *hardware* desenvolvido.

Na Tab. (2), demonstra-se a utilização dos recursos do FPGA. Pode-se observar que foram utilizados aproximadamente 7% dos 69120 registradores disponíveis. Também, apenas 4% da quantidade do total de BRAMs foi utilizada.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	5216	69120	7%
Number of Slice LUTs	4780	69120	6%
Number of fully used LUT-FF pairs	2646	7350	36%
Number of bonded IOBs	43	640	6%
Number of Block RAM/FIFO	7	148	4%
Number of BUFG/BUFGCTRLs	1	32	3%

Tabela 2 – Utilização dos recursos do FPGA.

A análise de desempenho foi realizada por meio de comparações dos tempos de execução da implementação em *hardware*, implementada em FPGA, e da implementação em *software* MATLAB, executada em computador de arquitetura tradicional.

Por sua vez, os testes em MATLAB foram realizados em um computador de ponta, com processador Intel® Core™ i7-3630QM de quatro núcleos, oito *threads*, frequência

de clock de 2.4GHz e 8GB de memória RAM. Para a realização dos testes em MATLAB, os tempos de leitura de uma imagem do sistema operacional e de impressão da imagem resultante na tela não foram contabilizados.

Os testes de tempo de execução foram efetuados para diferentes tamanhos de imagens, em ambas as plataformas. Os resultados podem ser vistos na Tab. (3).

TAMANHO DE IMAGEM	TEMPO DE EXECUÇÃO EM MATLAB	TEMPO DE EXECUÇÃO EM FPGA
64 X 64	320us	41us
128 X 128	810us	165us
256 X 256	2900us	660us
512 X 512	10500us	2630us

Tabela 3 – Comparação de tempos de execução em FPGA e em MATLAB.

É possível observar-se uma melhoria significativa de performance da implementação em FPGA. A Fig. (42) traz um gráfico comparativo para os tempos de execução nas duas plataformas.

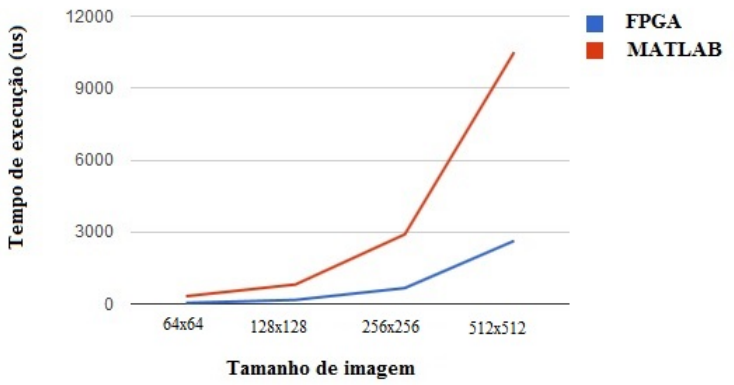


Figura 42 – Gráfico comparativo de performance entre implementações em FPGA e em MATLAB.

Uma das grandes vantagens da implementação em *hardware* é que o tempo de processamento é constante, dependendo do tamanho da imagem. O tempo de execução é dado pelo tempo necessário para ler a imagem e um período de latência do processamento, devido ao pipeline. Para a maioria dos casos, a latência é pequena se comparada com o carregamento de toda a imagem. Por isso, pode-se aproximar o tempo de execução ao tempo necessário para a leitura da imagem. Para imagens de tamanho 1024 x 1024, obteve-se uma taxa de processamento de até 95 quadros por segundo, com o FPGA operando em aproximadamente 100 MHz.

É notável a superioridade de desempenho da implementação em FPGA. Mesmo com uma frequência de clock aproximadamente 24 vezes menor, o *design* em FPGA obteve um aumento de velocidade da ordem de 500%, quando comparado à execução em *software*.

5 Conclusão

Atualmente, observa-se que o processamento de imagens tem sido aplicado num vasto campo, abrangendo as mais variadas atividades da tecnologia, desde o controle de qualidade em processos industriais, geoprocessamento e meteorologia a pesquisas em astrologia. Como exemplo, verifica-se, na área de Medicina, o uso do processamento de imagens como ferramenta imprescindível ao diagnóstico médico, por meio de técnicas avançadas como a tomografia computadorizada e a ressonância magnética.

A busca de soluções técnicas especializadas, torna-se, hoje em dia, fundamental para o alcance de resultados mais eficientes e eficazes, quando se pretende utilizar processamento de imagens.

Os requerimentos de performance de aplicações de processamento de imagens têm demandado, cada vez mais, um maior poder computacional, em especial, quanto a processamento em tempo real.

Neste caso, o processamento por hardware especializado surge como um elemento decisivo, pois possibilita uma redução do tempo de processamento em decorrência da execução paralela das operações.

Os sistemas de hardware reconfigurável possibilitam acelerar enormemente a execução de algoritmos, proporcionando uma alternativa de alto desempenho para soluções que antes eram executadas exclusivamente em software.

Neste trabalho, implementou-se em FPGA uma arquitetura de hardware dedicada à detecção de cantos em imagens, que é uma técnica essencial utilizada como etapa inicial em diversas aplicações. Sua execução eficiente, é portanto, imprescindível em aplicações em tempo real.

Descreveu-se, também, o processo de desenvolvimento, bem como as ferramentas necessárias para implementação do projeto.

A partir de comparações realizadas entre o processamento em FPGA e em arquitetura computacional convencional, a implementação em hardware demonstrou um ganho de performance da ordem de 500%.

Dessa forma, os resultados obtidos demonstraram como um hardware dedicado pode trazer ganhos de eficiência e performance a aplicações de processamento de imagens, utilizando-se técnicas de paralelismo e arquiteturas em pipeline.

Por fim, deseja-se que a realização deste trabalho, o qual apresentou resultados extremamente satisfatórios, possa servir de referencial aos estudiosos do tema, para o

desenvolvimento de novos projetos, na busca incessante de tecnologias de ponta, para que, constantemente, se possa alcançar a excelência na área de processamento de imagens.

Referências

ACHARYA, T.; RAY, A. *Image Processing: Principles and Applications*. [S.l.]: Wiley, 2005. ISBN 9780471745785.

AIZAWA, K.; SAKAUE, K.; SUENAGA, Y. *Image Processing Technologies: Algorithms, Sensors, and Applications*. [S.l.]: Taylor & Francis, 2004. (Signal Processing and Communications). ISBN 9780824750572.

ALZAHRANI, F. M.; CHEN, T. A real-time edge detector: Algorithm and vlsi architecture. Department of Electrical Engineering, Colorado State University, Fort Collins, CO 80523, U.S.A., 1997.

ANNADURAI, S. *Fundamentals Of Digital Image Processing*. [S.l.]: Pearson Education, 2007. ISBN 9788177584790.

BAILEY, D. G. *Design for Embedded Image Processing on FPGAS*. Massey University, New Zealand: John Wiley and Sons (Asia) Pte Ltd, 2011.

BRADSKI, G.; KAEHLER, A. *Learning OpenCV: Computer Vision with the OpenCV Library*. 1st. ed. Cambridge, MA: O'Reilly Media, 2008.

CARRO, L. *Projeto e Prototipação de Sistemas Digitais*. Porto Alegre, RS: Editora Universidade/UFRGS, 2001.

CASTLEMAN, K. *Digital Image Processing*. 1st. ed. [S.l.]: Prentice-Hall, 1996.

CHEN, J. et al. The comparison and application of corner detection algorithms. *JOURNAL OF MULTIMEDIA*, Beiing Institute of Technology, Beijing, China, 2009.

COLLINS, R. *Lecture: Harris Corner Detector*. 2005. Disponível em: <http://www.cse-.psu.edu/~rcollins/CSE486/lecture06.pdf>.

D.PARKS; GRAVEL, J. Corner detection. Disponível em: <http://www.cim.mc-gill.ca-/dparks/CornerDetector/harris.ht>.

FILHO, O. M.; NETO, H. V. *Processamento Digital de Imagens*. 1nd. ed. Rio de Janeiro, RJ, Brasil: Editora Brasport, 1999.

GONZALEZ, R. C.; WOODS, R. E. *Digital Image Processing*. 2nd. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

Gonzalez, R. C.; Woods, R. E.; Eddins, S. L. *Digital Image Processing Using MATLAB*. 2. ed. [S.l.]: Gatesmark Publishing, 2009.

HARRIS, C.; STEPHENS, M. A Combined Corner and Edge Detection. In: *Proceedings of The Fourth Alvey Vision Conference*. [s.n.], 1988. p. 147–151. Disponível em: <http://www.csse.uwa.edu.au/\~pk/research/matlabfns/Spatial/Docs/Harris-/A_Combined_Corner_and_Edge_Detector.p>.

Hsiao, P.-Y.; Lu, C.-L.; Fu, L.-C. Multilayered image processing for multiscale harris corner detection in digital realization. *IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS*, Department of Electrical Engineering, Colorado State University, Fort Collins, CO 80523, U.S.A., 2010.

JAIN, A. K. *Fundamentals of digital image processing*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

JIAO, W. a. Y. F.; HE, G. n integrated feature based method for sub-pixel image matching. *JOURNAL OF MULTIMEDIA*, Beiing Institute of Technology, Beijing, China, 2009.

KIRSCH, R. Seac and the start of image processing at the national bureau of standards. *IEEE Annals of the History of Computing*, 1998.

LIPSETT, R.; SCHAEFER, C.; USSERY, C. e. *VHDL: Hardware Description and Design.* [S.l.]: Kluwer Academic Press, 1989.

MAINI, R.; AGGARWAL, H. Study and comparison of various image edge detection techniques. 2009.

MEIXEDO, J. M. R. *Metodologias de projecto de baixo consumo para implementações em FPGA*. 2008.

MORAVEC, H. P. Visual mapping by a robot rover. *International Joint Conference on Artificial Intelligence*, 1979.

NAVABI, Z. *Digital Design and Implementation with Field Programmable Devices*. [S.l.]: Springer, 2005. ISBN 9781402080111.

Nixon, M. S.; Aguado, A. S. *Feature Extraction and Image Processing*. 2. ed. [S.l.]: Elsevier, 2008.

Rosten, E.; Drummond, T. Machine learning for high-speed corner detection. Department of Engineering, Cambridge University, UK, 2006.

SAMARAWICKRAMA, M. *Performance Evaluation of Vision Algorithms on FPGA*. [S.l.]: Universal Publishers, 2010. ISBN 9781599423739.

SASS, R.; SCHMIDT, A. *Embedded Systems Design with Platform FPGAs: Principles and Practices*. [S.l.]: Elsevier Science, 2010. ISBN 9780080921785.

SZELISKI, R. *Computer Vision: Algorithms and Applications*. 1st. ed. New York, NY, USA: Springer-Verlag New York, Inc., 2010.

WOODS, R. et al. *FPGA-based Implementation of Signal Processing Systems*. [S.l.]: Wiley, 2008. ISBN 9780470713778.

Anexos

ANEXO A – Código VHDL

Listing A.1 – Descrição top-level do hardware de Detector de Cantos de Harris

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4
5
6  -- Uncomment the following library declaration if using
7  -- arithmetic functions with Signed or Unsigned values
8  use IEEE.NUMERIC_STD.ALL;
9
10 use work.harris_package.all;
11
12 -- Uncomment the following library declaration if instantiating
13 -- any Xilinx primitives in this code.
14 --library UNISIM;
15 --use UNISIM.VComponents.all;
16
17
18 entity harris_corner_detector is
19     generic (
20                           IMAGE_STRIDE : integer:=256;
21                           I_WIDTH   : integer:=9;
22                           O_WIDTH   : integer:=30);
23     Port ( clk                                          : in std_logic;
24                           reset
                                        : in std_logic;
25                           pixel_in                                    : in
                                STD_LOGIC_VECTOR (I_WIDTH-1 downto 0);
26            pixel_in_valid                : in  STD_LOGIC;
27            corner_score                          : out  STD_LOGIC_VECTOR (
                  O_WIDTH-1 downto 0);
28            corner_score_valid     : out  STD_LOGIC);
29 end harris_corner_detector;
30
31 architecture Behavioral of harris_corner_detector is
32
33
34   --
```

```
35  -- Component Declaration
36  --
    ------------------------------------------------------------------------------

37
38          component neighborhood_extractor is
39      generic (
40                              IMAGE_STRIDE : integer;
41                              DATA_WIDTH   : integer;
42                              KERNEL_SIZE  : integer);
43      Port ( clk                          : in std_logic;
44                              reset                                   : in
                                    std_logic;
45                              data_in                 : in  STD_LOGIC_VECTOR (
                                    DATA_WIDTH-1 downto 0);
46             data_in_valid    : in  STD_LOGIC;
47             data_out                     : out  STD_LOGIC_VECTOR ( ((
                   KERNEL_SIZE**2)*DATA_WIDTH)-1 downto 0);
48             data_out_valid   : out  STD_LOGIC);
49          end component;
50
51          component dot_product is
52      generic (
53                              C_DATA_WIDTH     : integer;
54                              C_N_ELEMENTS     : integer;
55                              C_MULT_WIDTH     : integer;
56                              C_KERNEL_DATA_WIDTH : integer;
57                              C_OUTPUT_WIDTH   : integer);
58      Port ( clk                          : in  STD_LOGIC;
59             rst                          : in  STD_LOGIC;
60                              kernel                          : in  kernel_type;
61                              data_in                         : in
                                    STD_LOGIC_VECTOR ( C_N_ELEMENTS*C_DATA_WIDTH-1
                                      downto 0);
62                              data_in_valid    : in  STD_LOGIC;
63                              data_out                        : out
                                    STD_LOGIC_VECTOR ( C_OUTPUT_WIDTH-1 downto 0);
64             data_out_valid   : out  STD_LOGIC);
65           end component;
66
67  --
    ------------------------------------------------------------------------------

68  -- Internal signals
69  --
    ------------------------------------------------------------------------------
```

```
70
71                  signal sobel_neighborhood: STD_LOGIC_VECTOR ( ((
                        SOBEL_KERNEL_SIZE**2)*I_WIDTH)-1 downto 0);
72                  signal sobel_neighborhood_valid  : std_logic;
73
74                  signal Ix,Iy: STD_LOGIC_VECTOR ( SOBEL_OUTPUT_WIDTH-1
                        downto 0);
75                  signal Ix_valid,Iy_valid: STD_LOGIC;
76
77              signal Ix2,Iy2,Ixy: STD_LOGIC_VECTOR ( PRODUCT_SIZE-1 downto 0);
78                  signal concatIx2Iy2Ixy: STD_LOGIC_VECTOR ( 3*PRODUCT_SIZE-1
                        downto 0);
79                  signal en_gauss: STD_LOGIC;
80
81                  signal gauss_neighborhood: STD_LOGIC_VECTOR ( ((
                        GAUSS_KERNEL_SIZE**2)*3*PRODUCT_SIZE)-1 downto 0);
82                  signal gauss_neighborhood_valid  : std_logic;
83
84                  signal window_Ix2,window_Iy2,window_Ixy: STD_LOGIC_VECTOR(
                        ((GAUSS_KERNEL_SIZE**2)*PRODUCT_SIZE)-1 downto 0);
85                  signal Gx2_valid,Gy2_valid,Gxy_valid: STD_LOGIC;
86
87                  signal Gx2,Gy2,Gxy: STD_LOGIC_VECTOR ( GAUSS_OUTPUT_WIDTH-1
                        downto 0);
88
89
90                  signal  Gx2multGy2                                      :
                        STD_LOGIC_VECTOR ( 2*GAUSS_OUTPUT_WIDTH-1 downto 0);
91                  signal  GxymultGxy                                      :
                        STD_LOGIC_VECTOR ( 2*GAUSS_OUTPUT_WIDTH-1 downto 0);
92                  signal  Gx2addGy2                                       :
                        STD_LOGIC_VECTOR ( GAUSS_OUTPUT_WIDTH-1 downto 0);
93                  signal  Gx2multGy2_sub_GxymultGxy : STD_LOGIC_VECTOR ( 2*
                        GAUSS_OUTPUT_WIDTH-1 downto 0);
94                  signal  Gx2addGy2_2                                     :
                        STD_LOGIC_VECTOR ( 2*GAUSS_OUTPUT_WIDTH-1 downto 0);
95                  signal  R
                                          : STD_LOGIC_VECTOR ( 2*
                        GAUSS_OUTPUT_WIDTH-1 downto 0);
96
97  begin
98
99
100 --
```

```
101   -- Data path
```

```
102 --
    ------------------------------------------------------------------------------

103
104         NEIGHBORHOOD_SOBEL: neighborhood_extractor
105     generic map
106                                         (IMAGE_STRIDE     => IMAGE_STRIDE,
107                                               DATA_WIDTH        => I_WIDTH,
108                                     KERNEL_SIZE       => SOBEL_KERNEL_SIZE)
109           port map
110           (       clk => clk,
111                                       reset => reset,
112                                       data_in => pixel_in,
113                                       data_in_valid => pixel_in_valid,
114                                       data_out => sobel_neighborhood,
115                                       data_out_valid => sobel_neighborhood_valid
116                                       );
117
118         SOBEL_X: dot_product
119     generic map
120                                         (C_DATA_WIDTH     => I_WIDTH,
121                                               C_N_ELEMENTS      =>
                                                      SOBEL_KERNEL_SIZE**2,
122                                     C_MULT_WIDTH      => SOBEL_MULT_WIDTH,
123                                               C_KERNEL_DATA_WIDTH =>
                                                      SOBEL_KERNEL_DATA_WIDTH,
124                                               C_OUTPUT_WIDTH    =>
                                                      SOBEL_OUTPUT_WIDTH)
125           port map
126           (       clk => clk,
127                                       rst => reset,
128                                       kernel => sobel_x_kernel,
129                                       data_in => sobel_neighborhood,
130                                       data_in_valid => sobel_neighborhood_valid,
131                                       data_out => Ix,
132                                       data_out_valid => Ix_valid
133                                       );
134
135         SOBEL_Y: dot_product
136     generic map
137                                         (C_DATA_WIDTH     => I_WIDTH,
138                                               C_N_ELEMENTS      =>
                                                      SOBEL_KERNEL_SIZE**2,
139                                     C_MULT_WIDTH      => SOBEL_MULT_WIDTH,
140                                               C_KERNEL_DATA_WIDTH =>
                                                      SOBEL_KERNEL_DATA_WIDTH,
141                                               C_OUTPUT_WIDTH    =>
```

```
                                                    SOBEL_OUTPUT_WIDTH)
142          port map
143           (      clk => clk,
144                                  rst => reset,
145                                  kernel => sobel_y_kernel,
146                                  data_in => sobel_neighborhood,
147                                  data_in_valid => sobel_neighborhood_valid,
148                                  data_out => Iy,
149                                  data_out_valid => Iy_valid
150                                  );
151
152
153
154 DERIVATIVES_PRODUCTS: process(clk)
155 begin
156         if rising_edge(clk) then
157                 if reset = '1' then
158                         Ix2 <= (others => '0');
159                         Iy2 <= (others => '0');
160                         Ixy <=  (others => '0');
161                         en_gauss <= '0';
162                 else
163                         if(Ix_valid = '1') then
164
165                                 Ix2 <=  std_logic_vector(signed(Ix)*signed(
                                        Ix));
166                                 Iy2 <=  std_logic_vector(signed(Iy)*signed(
                                        Iy));
167                                 Ixy <= std_logic_vector(signed(Ix)*signed(
                                        Iy));
168
169                                 en_gauss <= '1';
170                         else
171                         Ix2 <=  (others => '0');
172                         Iy2 <= (others => '0');
173                         Ixy <=  (others => '0');
174                         en_gauss <= '0';
175                         end if;
176                 end if;
177         end if;
178 end process;
179
180
181         concatIx2Iy2Ixy <= (Ix2&Iy2&Ixy);
182
183         NEIGHBORHOOD_GAUSS: neighborhood_extractor
184     generic map
```

```
185                                     (IMAGE_STRIDE     => IMAGE_STRIDE-
                                            SOBEL_KERNEL_SIZE+1,
186                                         DATA_WIDTH        => (3*
                                                PRODUCT_SIZE),
187                                 KERNEL_SIZE       => GAUSS_KERNEL_SIZE)
188         port map
189          (      clk => clk,
190                                 reset => reset,
191                                 data_in => concatIx2Iy2Ixy,
192                                 data_in_valid => en_gauss,
193                                 data_out => gauss_neighborhood,
194                                 data_out_valid => gauss_neighborhood_valid
195                                 );
196
197
198 SEPARATE_PRODUCTS:
199  process (gauss_neighborhood)
200   begin
201
202                 for i in GAUSS_KERNEL_SIZE**2 downto 1 loop
203                         window_Ix2( (PRODUCT_SIZE*i-1) downto (PRODUCT_SIZE
                                *i-PRODUCT_SIZE)) <= gauss_neighborhood( (3*
                                PRODUCT_SIZE*i-1) downto (3*PRODUCT_SIZE*i-
                                PRODUCT_SIZE));
204                         window_Iy2( (PRODUCT_SIZE*i-1) downto (PRODUCT_SIZE
                                *i-PRODUCT_SIZE)) <= gauss_neighborhood( (3*
                                PRODUCT_SIZE*i-PRODUCT_SIZE-1) downto (3*
                                PRODUCT_SIZE*i-2*PRODUCT_SIZE));
205                         window_Ixy( (PRODUCT_SIZE*i-1) downto (PRODUCT_SIZE
                                *i-PRODUCT_SIZE)) <= gauss_neighborhood( (3*
                                PRODUCT_SIZE*i-2*PRODUCT_SIZE-1) downto (3*
                                PRODUCT_SIZE*i-3*PRODUCT_SIZE));
206                 end loop;
207
208   end process SEPARATE_PRODUCTS;
209
210         Gx2_gauss:dot_product
211     generic map
212                                     (C_DATA_WIDTH     => PRODUCT_SIZE,
213                                         C_N_ELEMENTS      =>
                                                GAUSS_KERNEL_SIZE**2,
214                                 C_MULT_WIDTH      => GAUSS_MULT_WIDTH,
215                                         C_KERNEL_DATA_WIDTH =>
                                                GAUSS_KERNEL_DATA_WIDTH,
216                                         C_OUTPUT_WIDTH    =>
                                                GAUSS_OUTPUT_WIDTH)
217         port map
```

```
218             (       clk => clk,
219                                     rst => reset,
220                                     kernel => gauss_kernel,
221                                     data_in => window_Ix2,
222                                     data_in_valid => gauss_neighborhood_valid,
223                                     data_out => Gx2,
224                                     data_out_valid => Gx2_valid
225                                     );
226
227
228         Gy2_gauss:dot_product
229     generic map
230                                       (C_DATA_WIDTH     => PRODUCT_SIZE,
231                                             C_N_ELEMENTS        =>
                                                GAUSS_KERNEL_SIZE**2,
232                                     C_MULT_WIDTH      => GAUSS_MULT_WIDTH,
233                                             C_KERNEL_DATA_WIDTH =>
                                                GAUSS_KERNEL_DATA_WIDTH,
234                                             C_OUTPUT_WIDTH      =>
                                                GAUSS_OUTPUT_WIDTH)
235         port map
236             (       clk => clk,
237                                     rst => reset,
238                                     kernel => gauss_kernel,
239                                     data_in => window_Iy2,
240                                     data_in_valid => gauss_neighborhood_valid,
241                                     data_out => Gy2,
242                                     data_out_valid => Gy2_valid
243                                     );
244
245         Gxy_gauss:dot_product
246     generic map
247                                       (C_DATA_WIDTH     => PRODUCT_SIZE,
248                                             C_N_ELEMENTS        =>
                                                GAUSS_KERNEL_SIZE**2,
249                                     C_MULT_WIDTH      => GAUSS_MULT_WIDTH,
250                                             C_KERNEL_DATA_WIDTH =>
                                                GAUSS_KERNEL_DATA_WIDTH,
251                                             C_OUTPUT_WIDTH      =>
                                                GAUSS_OUTPUT_WIDTH)
252         port map
253             (       clk => clk,
254                                     rst => reset,
255                                     kernel => gauss_kernel,
256                                     data_in => window_Ixy,
257                                     data_in_valid => gauss_neighborhood_valid,
258                                     data_out => Gxy,
```

```
259                                 data_out_valid => Gxy_valid
260                                 );
261
262
263 HARRIS_CORNER_SCORE: process(clk)
264
265                                 constant pipeline_stages : integer := 3;
266                             variable pipeline_delay : std_logic_vector(
                                    pipeline_stages-2 downto 0);
267
268 begin
269         if rising_edge(clk) then
270                 if reset = '1' then
271
272                         Gx2multGy2 <= (others => '0');
273                         GxymultGxy <= (others => '0');
274                         Gx2addGy2 <= (others => '0');
275                         Gx2multGy2_sub_GxymultGxy <= (others => '0');
276                         Gx2addGy2_2 <= (others => '0');
277                         R <= (others => '0');
278
279                         corner_score_valid <= '0';
280                         pipeline_delay := (others => '0');
281
282                 else
283
284                             corner_score_valid <= pipeline_delay(
                                    pipeline_delay'high);
285                             pipeline_delay := pipeline_delay(pipeline_delay'
                                    high-1 downto 0)&Gx2_valid;
286
287                                 Gx2multGy2 <=  std_logic_vector(signed(Gx2)
                                        *signed(Gy2));
288                                 GxymultGxy <=  std_logic_vector(signed(Gxy)
                                        *signed(Gxy));
289                                 Gx2addGy2 <=  std_logic_vector(signed(Gx2)+
                                        signed(Gy2));
290
291                                 Gx2multGy2_sub_GxymultGxy <=
                                        std_logic_vector(signed(Gx2multGy2)-
                                        signed(GxymultGxy));
292                                 Gx2addGy2_2 <=  std_logic_vector(signed(
                                        Gx2addGy2)*signed(Gx2addGy2));
293
294                                 R <= std_logic_vector(  signed(
                                        Gx2multGy2_sub_GxymultGxy)  -  signed(
                                        Gx2addGy2_2)/32     );
```

```
295
296                 end if;
297         end if;
298 end process;
299
300 corner_score <= R;
301
302 end Behavioral;
```

Listing A.2 – Hardware para envio de pixels ao detector

```
 1
 2 library IEEE;
 3 use IEEE.STD_LOGIC_1164.ALL;
 4
 5
 6 -- Uncomment the following library declaration if using
 7 -- arithmetic functions with Signed or Unsigned values
 8 use IEEE.NUMERIC_STD.ALL;
 9
10 use work.harris_package.all;
11
12 -- Uncomment the following library declaration if instantiating
13 -- any Xilinx primitives in this code.
14 --library UNISIM;
15 --use UNISIM.VComponents.all;
16
17
18 entity harris_corner_detector is
19     generic (
20                             IMAGE_STRIDE : integer:=256;
21                             I_WIDTH   : integer:=9;
22                             O_WIDTH   : integer:=30);
23     Port ( clk                                          : in std_logic;
24                             reset
                                        : in std_logic;
25                             pixel_in                                 : in
                                STD_LOGIC_VECTOR (I_WIDTH-1 downto 0);
26            pixel_in_valid                : in  STD_LOGIC;
27            corner_score                          : out  STD_LOGIC_VECTOR (
                O_WIDTH-1 downto 0);
28            corner_score_valid     : out  STD_LOGIC);
29 end harris_corner_detector;
30
31 architecture Behavioral of harris_corner_detector is
32
33
34   --
```

```
35  -- Component Declaration
36  --
37
38      component neighborhood_extractor is
39    generic (
40                          IMAGE_STRIDE : integer;
41                          DATA_WIDTH  : integer;
42                          KERNEL_SIZE  : integer);
43    Port ( clk                          : in std_logic;
44                          reset                                   : in
                                std_logic;
45                          data_in                     : in  STD_LOGIC_VECTOR (
                                DATA_WIDTH-1 downto 0);
46           data_in_valid    : in  STD_LOGIC;
47           data_out                       : out  STD_LOGIC_VECTOR ( ((
                KERNEL_SIZE**2)*DATA_WIDTH)-1 downto 0);
48           data_out_valid   : out  STD_LOGIC);
49        end component;
50
51        component dot_product is
52    generic (
53                          C_DATA_WIDTH     : integer;
54                          C_N_ELEMENTS     : integer;
55                          C_MULT_WIDTH     : integer;
56                          C_KERNEL_DATA_WIDTH : integer;
57                          C_OUTPUT_WIDTH  : integer);
58    Port ( clk                          : in  STD_LOGIC;
59           rst                          : in  STD_LOGIC;
60                          kernel                          : in  kernel_type;
61                          data_in                         : in
                               STD_LOGIC_VECTOR ( C_N_ELEMENTS*C_DATA_WIDTH-1
                                downto 0);
62                          data_in_valid    : in  STD_LOGIC;
63                          data_out                        : out
                               STD_LOGIC_VECTOR ( C_OUTPUT_WIDTH-1 downto 0);
64           data_out_valid  : out  STD_LOGIC);
65         end component;
66
67  --
68  -- Internal signals
69  --
```

```
70
71                  signal sobel_neighborhood: STD_LOGIC_VECTOR ( ((
                        SOBEL_KERNEL_SIZE**2)*I_WIDTH)-1 downto 0);
72                  signal sobel_neighborhood_valid  : std_logic;
73
74                  signal Ix,Iy: STD_LOGIC_VECTOR ( SOBEL_OUTPUT_WIDTH-1
                        downto 0);
75                  signal Ix_valid,Iy_valid: STD_LOGIC;
76
77              signal Ix2,Iy2,Ixy: STD_LOGIC_VECTOR ( PRODUCT_SIZE-1 downto 0);
78                  signal concatIx2Iy2Ixy: STD_LOGIC_VECTOR ( 3*PRODUCT_SIZE-1
                         downto 0);
79                  signal en_gauss: STD_LOGIC;
80
81                  signal gauss_neighborhood: STD_LOGIC_VECTOR ( ((
                        GAUSS_KERNEL_SIZE**2)*3*PRODUCT_SIZE)-1 downto 0);
82                  signal gauss_neighborhood_valid  : std_logic;
83
84                  signal window_Ix2,window_Iy2,window_Ixy: STD_LOGIC_VECTOR(
                        ((GAUSS_KERNEL_SIZE**2)*PRODUCT_SIZE)-1 downto 0);
85                  signal Gx2_valid,Gy2_valid,Gxy_valid: STD_LOGIC;
86
87                  signal Gx2,Gy2,Gxy: STD_LOGIC_VECTOR ( GAUSS_OUTPUT_WIDTH-1
                         downto 0);
88
89
90                  signal  Gx2multGy2                                          :
                         STD_LOGIC_VECTOR ( 2*GAUSS_OUTPUT_WIDTH-1 downto 0);
91                  signal  GxymultGxy                                          :
                         STD_LOGIC_VECTOR ( 2*GAUSS_OUTPUT_WIDTH-1 downto 0);
92                  signal  Gx2addGy2                                           :
                         STD_LOGIC_VECTOR ( GAUSS_OUTPUT_WIDTH-1 downto 0);
93                  signal  Gx2multGy2_sub_GxymultGxy : STD_LOGIC_VECTOR ( 2*
                        GAUSS_OUTPUT_WIDTH-1 downto 0);
94                  signal  Gx2addGy2_2                                         :
                        STD_LOGIC_VECTOR ( 2*GAUSS_OUTPUT_WIDTH-1 downto 0);
95                  signal  R
                                                : STD_LOGIC_VECTOR ( 2*
                        GAUSS_OUTPUT_WIDTH-1 downto 0);
96
97  begin
98
99
100  --
```

```
101   -- Data path
102   --
      ------------------------------------------------------------------------------

103
104       NEIGHBORHOOD_SOBEL: neighborhood_extractor
105     generic map
106                                       (IMAGE_STRIDE     => IMAGE_STRIDE,
107                                             DATA_WIDTH          => I_WIDTH,
108                                   KERNEL_SIZE       => SOBEL_KERNEL_SIZE)
109           port map
110           (       clk => clk,
111                                     reset => reset,
112                                     data_in => pixel_in,
113                                     data_in_valid => pixel_in_valid,
114                                     data_out => sobel_neighborhood,
115                                     data_out_valid => sobel_neighborhood_valid
116                                     );
117
118       SOBEL_X: dot_product
119     generic map
120                                       (C_DATA_WIDTH     => I_WIDTH,
121                                             C_N_ELEMENTS        =>
                                                SOBEL_KERNEL_SIZE**2,
122                                   C_MULT_WIDTH      => SOBEL_MULT_WIDTH,
123                                             C_KERNEL_DATA_WIDTH =>
                                                SOBEL_KERNEL_DATA_WIDTH,
124                                             C_OUTPUT_WIDTH      =>
                                                SOBEL_OUTPUT_WIDTH)
125           port map
126           (       clk => clk,
127                                     rst => reset,
128                                     kernel => sobel_x_kernel,
129                                     data_in => sobel_neighborhood,
130                                     data_in_valid => sobel_neighborhood_valid,
131                                     data_out => Ix,
132                                     data_out_valid => Ix_valid
133                                     );
134
135       SOBEL_Y: dot_product
136     generic map
137                                       (C_DATA_WIDTH     => I_WIDTH,
138                                             C_N_ELEMENTS        =>
                                                SOBEL_KERNEL_SIZE**2,
139                                   C_MULT_WIDTH      => SOBEL_MULT_WIDTH,
140                                             C_KERNEL_DATA_WIDTH =>
```

```
                                                    SOBEL_KERNEL_DATA_WIDTH,
141                                                 C_OUTPUT_WIDTH     =>
                                                    SOBEL_OUTPUT_WIDTH)
142          port map
143           (      clk => clk,
144                                  rst => reset,
145                                  kernel => sobel_y_kernel,
146                                  data_in => sobel_neighborhood,
147                                  data_in_valid => sobel_neighborhood_valid,
148                                  data_out => Iy,
149                                  data_out_valid => Iy_valid
150                                  );
151
152
153
154 DERIVATIVES_PRODUCTS: process(clk)
155 begin
156         if rising_edge(clk) then
157                 if reset = '1' then
158                         Ix2 <= (others => '0');
159                         Iy2 <= (others => '0');
160                         Ixy <=  (others => '0');
161                         en_gauss <= '0';
162                 else
163                         if(Ix_valid = '1') then
164
165                                 Ix2 <=  std_logic_vector(signed(Ix)*signed(
                                        Ix));
166                                 Iy2 <=  std_logic_vector(signed(Iy)*signed(
                                        Iy));
167                                 Ixy <= std_logic_vector(signed(Ix)*signed(
                                        Iy));
168
169                                 en_gauss <= '1';
170                         else
171                         Ix2 <=  (others => '0');
172                         Iy2 <= (others => '0');
173                         Ixy <=  (others => '0');
174                         en_gauss <= '0';
175                         end if;
176                 end if;
177         end if;
178 end process;
179
180
181         concatIx2Iy2Ixy <= (Ix2&Iy2&Ixy);
182
```

```
183         NEIGHBORHOOD_GAUSS: neighborhood_extractor
184     generic map
185                                     (IMAGE_STRIDE     => IMAGE_STRIDE-
                                            SOBEL_KERNEL_SIZE+1,
186                                         DATA_WIDTH        => (3*
                                                PRODUCT_SIZE),
187                                 KERNEL_SIZE       => GAUSS_KERNEL_SIZE)
188           port map
189            (      clk => clk,
190                                   reset => reset,
191                                   data_in => concatIx2Iy2Ixy,
192                                   data_in_valid => en_gauss,
193                                   data_out => gauss_neighborhood,
194                                   data_out_valid => gauss_neighborhood_valid
195                                   );
196
197
198 SEPARATE_PRODUCTS:
199  process (gauss_neighborhood)
200   begin
201
202                 for i in GAUSS_KERNEL_SIZE**2 downto 1 loop
203                         window_Ix2( (PRODUCT_SIZE*i-1) downto (PRODUCT_SIZE
                                *i-PRODUCT_SIZE)) <= gauss_neighborhood( (3*
                                PRODUCT_SIZE*i-1) downto (3*PRODUCT_SIZE*i-
                                PRODUCT_SIZE));
204                         window_Iy2( (PRODUCT_SIZE*i-1) downto (PRODUCT_SIZE
                                *i-PRODUCT_SIZE)) <= gauss_neighborhood( (3*
                                PRODUCT_SIZE*i-PRODUCT_SIZE-1) downto (3*
                                PRODUCT_SIZE*i-2*PRODUCT_SIZE));
205                         window_Ixy( (PRODUCT_SIZE*i-1) downto (PRODUCT_SIZE
                                *i-PRODUCT_SIZE)) <= gauss_neighborhood( (3*
                                PRODUCT_SIZE*i-2*PRODUCT_SIZE-1) downto (3*
                                PRODUCT_SIZE*i-3*PRODUCT_SIZE));
206                 end loop;
207
208   end process SEPARATE_PRODUCTS;
209
210         Gx2_gauss:dot_product
211     generic map
212                                     (C_DATA_WIDTH     => PRODUCT_SIZE,
213                                         C_N_ELEMENTS      =>
                                                GAUSS_KERNEL_SIZE**2,
214                                 C_MULT_WIDTH      => GAUSS_MULT_WIDTH,
215                                         C_KERNEL_DATA_WIDTH =>
                                                GAUSS_KERNEL_DATA_WIDTH,
216                                         C_OUTPUT_WIDTH    =>
```

```
                                                GAUSS_OUTPUT_WIDTH)
217           port map
218           (       clk => clk,
219                                   rst => reset,
220                                   kernel => gauss_kernel,
221                                   data_in => window_Ix2,
222                                   data_in_valid => gauss_neighborhood_valid,
223                                   data_out => Gx2,
224                                   data_out_valid => Gx2_valid
225                                   );
226
227
228           Gy2_gauss:dot_product
229       generic map
230                                     (C_DATA_WIDTH     => PRODUCT_SIZE,
231                                           C_N_ELEMENTS        =>
                                                GAUSS_KERNEL_SIZE**2,
232                                    C_MULT_WIDTH       => GAUSS_MULT_WIDTH,
233                                           C_KERNEL_DATA_WIDTH =>
                                                GAUSS_KERNEL_DATA_WIDTH,
234                                           C_OUTPUT_WIDTH      =>
                                                GAUSS_OUTPUT_WIDTH)
235           port map
236           (       clk => clk,
237                                   rst => reset,
238                                   kernel => gauss_kernel,
239                                   data_in => window_Iy2,
240                                   data_in_valid => gauss_neighborhood_valid,
241                                   data_out => Gy2,
242                                   data_out_valid => Gy2_valid
243                                   );
244
245           Gxy_gauss:dot_product
246       generic map
247                                     (C_DATA_WIDTH     => PRODUCT_SIZE,
248                                           C_N_ELEMENTS        =>
                                                GAUSS_KERNEL_SIZE**2,
249                                    C_MULT_WIDTH       => GAUSS_MULT_WIDTH,
250                                           C_KERNEL_DATA_WIDTH =>
                                                GAUSS_KERNEL_DATA_WIDTH,
251                                           C_OUTPUT_WIDTH      =>
                                                GAUSS_OUTPUT_WIDTH)
252           port map
253           (       clk => clk,
254                                   rst => reset,
255                                   kernel => gauss_kernel,
256                                   data_in => window_Ixy,
```

```
257                                 data_in_valid => gauss_neighborhood_valid,
258                                 data_out => Gxy,
259                                 data_out_valid => Gxy_valid
260                                 );
261
262
263 HARRIS_CORNER_SCORE: process(clk)
264
265                                 constant pipeline_stages : integer := 3;
266                             variable pipeline_delay : std_logic_vector(
                                    pipeline_stages-2 downto 0);
267
268 begin
269         if rising_edge(clk) then
270                 if reset = '1' then
271
272                         Gx2multGy2 <= (others => '0');
273                         GxymultGxy <= (others => '0');
274                         Gx2addGy2 <= (others => '0');
275                         Gx2multGy2_sub_GxymultGxy <= (others => '0');
276                         Gx2addGy2_2 <= (others => '0');
277                         R <= (others => '0');
278
279                         corner_score_valid <= '0';
280                         pipeline_delay := (others => '0');
281
282                 else
283
284                             corner_score_valid <= pipeline_delay(
                                    pipeline_delay'high);
285                             pipeline_delay := pipeline_delay(pipeline_delay'
                                    high-1 downto 0)&Gx2_valid;
286
287                                 Gx2multGy2 <=  std_logic_vector(signed(Gx2)
                                        *signed(Gy2));
288                                 GxymultGxy <=  std_logic_vector(signed(Gxy)
                                        *signed(Gxy));
289                                 Gx2addGy2 <=  std_logic_vector(signed(Gx2)+
                                        signed(Gy2));
290
291                                 Gx2multGy2_sub_GxymultGxy <=
                                        std_logic_vector(signed(Gx2multGy2)-
                                        signed(GxymultGxy));
292                                 Gx2addGy2_2 <=  std_logic_vector(signed(
                                        Gx2addGy2)*signed(Gx2addGy2));
293
294                                 R <= std_logic_vector(  signed(
```

```
                                        Gx2multGy2_sub_GxymultGxy) - signed(
                                        Gx2addGy2_2)/32      );
295
296                 end if;
297         end if;
298 end process;
299
300 corner_score <= R;
301
302 end Behavioral;
```

Listing A.3 – Módulo de obtenção de vizinhança

```
 1
 2 library IEEE;
 3 use IEEE.STD_LOGIC_1164.ALL;
 4 use work.util.log2ceil;
 5 --use IEEE.NUMERIC_STD.ALL;
 6
 7
 8 entity neighborhood_extractor is
 9     generic (
10                           IMAGE_STRIDE : integer;
11                           DATA_WIDTH   : integer;                    ---8 bit
                                 - 1 Pixel
12                           KERNEL_SIZE  : integer);
13     Port ( clk                          : in std_logic;
14                           reset                                  : in
                                 std_logic;
15                           data_in                    : in  STD_LOGIC_VECTOR (
                                 DATA_WIDTH-1 downto 0);
16            data_in_valid    : in  STD_LOGIC;
17            data_out                   : out  STD_LOGIC_VECTOR ( ((
                  KERNEL_SIZE**2)*DATA_WIDTH)-1 downto 0);
18            data_out_valid   : out  STD_LOGIC);
19 end neighborhood_extractor;
20
21 architecture Behavioral of neighborhood_extractor is
22
23
24   --
       ------------------------------------------------------------------------

25   -- Component Declaration
26   --
       ------------------------------------------------------------------------

27
```

```
28 component fifo is
29   generic (
30         ADDRESS_WIDTH : integer:=8;   ---8 bit
31         DATA_WIDTH : integer:=8);     ---8 bit
32   port (
33     reset, clk, r, w : in  std_logic;
34     empty, full      : out std_logic;
35     d                : in  std_logic_vector(DATA_WIDTH-1 downto 0);
36     q                : out std_logic_vector(DATA_WIDTH-1 downto 0));
37 end component;
38
39   --
       ------------------------------------------------------------------------

40   -- Internal signals
41   --
       ------------------------------------------------------------------------

42
43
44
45   type line_std_logic is  array(KERNEL_SIZE-1 downto 0) of std_logic;
46   signal rw,empty,full : line_std_logic;
47
48   type line_in_out is  array(KERNEL_SIZE-1 downto 0) of std_logic_vector(
       DATA_WIDTH-1 downto 0);
49   signal data_line: line_in_out;
50
51    signal send_data_out_valid: std_logic := '0';
52         signal eol_count: integer range  0 to IMAGE_STRIDE-1:=0;
53         signal pixel_count: integer range  0 to IMAGE_STRIDE-1:=0;
54
55         -- States for the control logic
56         type   STATE_TYPE is ( WAIT_FIRST_CONV_BLOCK,WAIT_COMPLETE_LINE,
             WAIT_FIRST_PIXELS_LINE);
57         signal state     : STATE_TYPE := WAIT_FIRST_CONV_BLOCK;
58
59  begin
60
61   --
       ------------------------------------------------------------------------

62   -- Data path
63   --
       ------------------------------------------------------------------------

64
```

```
65   INST_FIFOS:
66                   for i in KERNEL_SIZE-2 downto 0 generate
67                   LINE_BUFFER : fifo
68                   generic map( ADDRESS_WIDTH => log2ceil(IMAGE_STRIDE),
69                                                 DATA_WIDTH      =>
                                                      DATA_WIDTH)
70                   port map(reset => reset,
71                                          clk    => clk,
72                                          r      => rw(i+1),
73                                          w                => rw(i),
74                                          empty  => empty(i),
75                                          full   => full(i),
76                                          d      => data_line(i),
77                                          q      => data_line(i+1)
78                                          );
79   end generate INST_FIFOS;
80
81
82 --
     ------------------------------------------------------------------------------

83 -- Control path
84 --
     ------------------------------------------------------------------------------

85
86  SHIFT_CONV_REGISTERS:
87  process (clk)
88
89         -- Registers for convolution
90   type conv is array(0 to KERNEL_SIZE-1) of std_logic_vector(DATA_WIDTH*
       KERNEL_SIZE-1 downto 0);
91   variable conv_block    : conv;
92
93   begin
94                   if rising_edge(clk) then
95                           if reset= '1' then
96
97                                   for i in  KERNEL_SIZE-1 downto 0 loop
98                                           conv_block(i):= (others => '0');
99                                   end loop;
100
101                          else
102
103                                  if(rw(KERNEL_SIZE-1) = '1') then
104
105                                          for i in  KERNEL_SIZE-1 downto 0
```

```
                                                    loop
106                                                     conv_block(i):=data_line(i)
                                                            &conv_block(i)((
                                                            DATA_WIDTH*KERNEL_SIZE
                                                            -1) downto DATA_WIDTH);
107                                                 end loop;
108                                         else
109                                                 for i in  KERNEL_SIZE-1 downto 0
                                                        loop
110                                                     conv_block(i):=conv_block(i
                                                            );
111                                                 end loop;
112                                         end if;
113                                 end if;
114
115                         for i in KERNEL_SIZE downto 1 loop
116                                 data_out((DATA_WIDTH*KERNEL_SIZE*i-1) downto (
                                        DATA_WIDTH*KERNEL_SIZE*i - DATA_WIDTH*
                                        KERNEL_SIZE)) <= conv_block(KERNEL_SIZE-i);
117                         end loop;
118
119                         end if;
120   end process SHIFT_CONV_REGISTERS;
121
122
123  LINES_RW:
124  process (clk)
125   begin
126
127                         if rising_edge(clk) then
128                                 if reset= '1' then
129
130                                         data_line(0) <= (others =>'0');
131                                         for i in  KERNEL_SIZE-1 downto 0 loop
132                                                 rw(i) <= '0';
133                                         end loop;
134                                         eol_count<=0;
135                                         pixel_count<=0;
136
137                                 else
138                                     if(data_in_valid = '1') then
139
140                                                                 if pixel_count =
                                                                        IMAGE_STRIDE-1
                                                                        then
141                                                                     pixel_count
                                                                            <=0;
```

```
142                                                 eol_count
                                                        <=
                                                        eol_count
                                                        +1;
143                                             else
144                                                 pixel_count
                                                        <=
                                                        pixel_count
                                                        +1;
145                                             end if;
146
147                                             data_line(0) <=
                                                    data_in;
148                                             for i in
                                                    KERNEL_SIZE-1
                                                    downto 0 loop
149                                                 if(i <=
                                                        eol_count
                                                        ) then
150                                                     rw(
                                                            i
                                                            )

                                                            <=

                                                            '1';

151                                                 else
152                                                     rw(
                                                            i
                                                            )

                                                            <=

                                                            '0';

153                                                 end if;
154                                             end loop;
155                         else
156
157                                             data_line(0) <= (
                                                    others =>'0');
158                                             for i in
                                                    KERNEL_SIZE-1
                                                    downto 0 loop
159                                                 rw(i) <=
                                                        '0';
```

```
160                                                     end loop;
161                         end if;
162                 end if;
163                 end if;
164  end process LINES_RW;
165
166
167  DATA_OUT_VALID_PROCESS:
168  process (clk)
169   begin
170                 if rising_edge(clk) then
171                         if reset= '1' then
172
173                                         data_out_valid <= '0';
174
175                         else
176
177                                 if ((eol_count >= KERNEL_SIZE-1) and (
                                        pixel_count >= KERNEL_SIZE-1)) then
178                                         send_data_out_valid <= '1';
179                                 else
180                                         send_data_out_valid <= '0';
181                                 end if;
182
183                                 data_out_valid <= send_data_out_valid;
184
185                         end if;
186
187                 end if;
188   end process DATA_OUT_VALID_PROCESS;
189
190 end Behavioral;
```

Listing A.4 – Descrição de FIFO como sendo um buffer circular

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  --use ieee.std_logic_unsigned.all;
5
6
7  entity fifo is
8    generic (
9           ADDRESS_WIDTH : integer:=2;   ---8 bit
10          DATA_WIDTH : integer:=32);     ---8 bit
11   port (
12     reset, clk, r, w : in  std_logic;
13     empty, full      : out std_logic;
```

```
14      d                : in  std_logic_vector(DATA_WIDTH-1 downto 0);
15      q                : out std_logic_vector(DATA_WIDTH-1 downto 0));
16  end fifo;
17
18  architecture rtl of fifo is
19
20    signal   rcntr, wcntr        : unsigned(ADDRESS_WIDTH-1 downto 0);
21    type     regtype is array (0 to ((2**ADDRESS_WIDTH)-1)) of
          std_logic_vector(DATA_WIDTH-1 downto 0);
22    signal   reg                 : regtype;
23    signal   rw                  : std_logic_vector(1 downto 0);
24    signal   full_buf, empty_buf: std_logic;
25  begin
26    rw <= r & w;
27    seq : process(reset, clk)
28    begin
29
30      if rising_edge(clk) then
31            if reset = '1' then
32        rcntr     <= (others => '0');
33        wcntr     <= (others => '0');
34        empty_buf <= '1';
35        full_buf  <= '0';
36            else
37
38        case rw is
39          when "11" =>
40            -- read and write at the same time
41            rcntr                    <= rcntr + 1;
42            wcntr                    <= wcntr + 1;
43            reg(to_integer(wcntr)) <= d;
44          when "10" =>
45            -- only read
46              if (rcntr + 1) = wcntr then
47                empty_buf <= '1';
48              end if;
49              rcntr <= rcntr + 1;
50
51            full_buf <= '0';
52
53          when "01" =>
54            -- only write
55            reg(to_integer(wcntr)) <= d;
56              if (wcntr + 1) = rcntr then
57                full_buf <= '1';
58              end if;
59              wcntr <= wcntr + 1;
```

```
60                          empty_buf <= '0';
61          when others => null;
62        end case;
63            end if;
64      end if;
65    end process;
66
67    q      <= reg(to_integer(rcntr));
68
69    full  <= full_buf;
70    empty <= empty_buf;
71 end rtl;
```

Listing A.5 – Arquitetura de produto interno

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4  use IEEE.NUMERIC_STD.ALL;
5  use work.util.log2ceil;
6  use work.harris_package.all;
7
8  entity dot_product is
9      generic (
10                          C_DATA_WIDTH     : integer;
11                          C_N_ELEMENTS     : integer;
12                          C_MULT_WIDTH     : integer;
13                          C_KERNEL_DATA_WIDTH : integer;
14                          C_OUTPUT_WIDTH   : integer);
15     Port ( clk                          : in  STD_LOGIC;
16            rst                          : in  STD_LOGIC;
17                          kernel                       : in  kernel_type;
18                          data_in                      : in
                               STD_LOGIC_VECTOR ( C_N_ELEMENTS*C_DATA_WIDTH-1
                                downto 0);
19                          data_in_valid    : in  STD_LOGIC;
20                          data_out                     : out
                               STD_LOGIC_VECTOR ( C_OUTPUT_WIDTH-1 downto 0);
21            data_out_valid   : out  STD_LOGIC);
22 end dot_product;
23
24 architecture Behavioral of dot_product is
25
26
27   --
     ------------------------------------------------------------------------

28   -- Component Declaration
```

```
29    --
      ------------------------------------------------------------------------------

30          component adder_tree is
31          generic ( NINPUTS : integer;
32                                    IWIDTH   : integer;
33                                    OWIDTH   : integer);
34          port (rst     : in  std_logic;
35                  clk     : in  std_logic;
36                  en                :       in  STD_LOGIC;
37                          d       : in  std_logic_vector(NINPUTS*IWIDTH-1
                                downto 0);
38                          q       : out std_logic_vector(OWIDTH-1 downto 0);
39                          valid   : out STD_LOGIC);
40          end component;
41
42    --
      ------------------------------------------------------------------------------

43    -- Internal signals
44    --
      ------------------------------------------------------------------------------

45                  signal en_add: std_logic;
46                  signal mult_res : std_logic_vector(C_N_ELEMENTS*
                        C_MULT_WIDTH-1 downto 0);
47
48  begin
49
50
51    --
      ------------------------------------------------------------------------------

52    -- Data path
53    --
      ------------------------------------------------------------------------------

54
55  Parallel_product: process(clk)
56
57  begin
58          if rising_edge(clk) then
59                  if rst = '1' then
60                          mult_res <= (others => '0');
61                  else
62                          if(data_in_valid = '1') then
63
```

```
64                                  for i in C_N_ELEMENTS downto 1 loop
65                                          mult_res(   (C_MULT_WIDTH*i-1)
                                                downto (C_MULT_WIDTH*i-
                                                C_MULT_WIDTH)     ) <=
                                                std_logic_vector(signed(data_in
                                                ((C_DATA_WIDTH*i-1) downto (
                                                C_DATA_WIDTH*i-C_DATA_WIDTH))) *
                                                 to_signed(kernel(C_N_ELEMENTS-i
                                                ),C_KERNEL_DATA_WIDTH));
66                                  end loop;
67
68                                  en_add <= '1';
69
70                          else
71                                  mult_res <=mult_res;
72                                  en_add <= '0';
73
74                          end if;
75                  end if;
76          end if;
77  end process;
78
79
80
81          adder_tree_map : adder_tree
82          generic map(NINPUTS     => C_N_ELEMENTS,
83                                          IWIDTH      => C_MULT_WIDTH,
84                                  OWIDTH      => C_OUTPUT_WIDTH)
85          port map
86            (       clk => clk,
87                                  rst => rst,
88                                  en => en_add,
89                                  d => mult_res,
90                                  q => data_out,
91                                  valid => data_out_valid
92                                  );
93
94
95
96
97  end Behavioral;
```

Listing A.6 – Descrição de árvore de soma binária

```
1  --------------------------------------------------------------------------------

2
```

```
3
4  library ieee;
5  use ieee.std_logic_1164.all;
6  use ieee.numeric_std.all;
7  use work.util.log2ceil;
8
9  entity adder_tree is
10         generic (
11                 -- Number of inputs
12                 NINPUTS : integer;
13                 -- Input data width
14                 IWIDTH  : integer;
15                 -- Output data width
16                 -- * full-precision requires that
17                 --   OWIDTH >= IWIDTH + ceil(log2(NINPUTS))
18                 OWIDTH  : integer
19         );
20         port (
21                 -- Reset, clock and enable
22                 rst     : in  std_logic;
23                 clk     : in  std_logic;
24                 en              :       in  STD_LOGIC;
25                 -- Input data
26                 d       : in  std_logic_vector(NINPUTS*IWIDTH-1 downto 0);
27                 -- Output data
28                 q       : out std_logic_vector(OWIDTH-1 downto 0);
29                 valid           :       out  STD_LOGIC
30         );
31 end entity;
32
33 -- ------------------------------------------------------------------------
34
35 architecture Behavioral of adder_tree is
36
37         -- ------------------------------------------------------------------
38         -- Local functions
39         -- ------------------------------------------------------------------
40         --
41         -- Input lookup and conversion
42         impure function din(i : integer) return signed is
43         begin
44
45                 return signed(d((i+1)*IWIDTH-1 downto i*IWIDTH));
46
47         end function;
48
49         -- ------------------------------------------------------------------
```

```
50          -- Constants
51          -- ------------------------------------------------------------------------
52          --
53          -- Number of stages required to sum the inputs
54          constant NSTAGES : integer := log2ceil(NINPUTS);
55          -- Number of inputs padded to next power-of-2
56          -- * the width of the first stage of sums
57          constant PWIDTH : integer := 2**NSTAGES;
58          -- The total number of sums in the pipeline
59          constant NSUMS : integer := PWIDTH-1;
60          -- The width of the last sum (used for all sums)
61          constant SWIDTH : integer := IWIDTH + NSTAGES;
62
63          -- ------------------------------------------------------------------------
64          -- Signals
65          -- ------------------------------------------------------------------------
66          --
67          -- Sum pipeline
68          type sum_t is array (0 to NSUMS-1) of signed(SWIDTH-1 downto 0);
69          signal sum : sum_t;
70
71  begin
72
73
74          -- ------------------------------------------------------------------------
75          -- Pipelined adders
76          -- ------------------------------------------------------------------------
77          --
78          process(clk,rst)
79                  variable prev_index : integer;
80                  variable curr_index : integer;
81                  variable num_sums   : integer;
82
83                  variable en_delay : std_logic_vector(NSTAGES-2 downto 0);
84
85          begin
86                  if rising_edge(clk) then
87                          if (rst = '1') then
88                        sum <= (others => (others => '0'));
89                                  en_delay := (others => '0');
90                                  valid <= '0';
91                          else
92                            valid <= en_delay(en_delay'high);
93                            en_delay := en_delay(en_delay'high-1 downto 0)&en
                                  ;
94
95                                  -- First stage; pairs of input sums
```

```
96                                  -- * unused power-of-2 padding is left at
                                        the
97                                  --    reset value of zero.
98                                  num_sums := PWIDTH/2;
99                                  for j in 0 to num_sums-1 loop
100                                         if (2*j+1 < NINPUTS) then
101                                                 -- Resize and then sum (to
                                                        avoid overflow)
102                                                 sum(j) <= resize(din(2*j),
                                                        SWIDTH) + resize(din(2*j
                                                        +1), SWIDTH);
103                                         elsif (2*j < NINPUTS) then
104                                                 sum(j) <= resize(din(2*j),
                                                        SWIDTH);
105                                         end if;
106                                 end loop;
107
108                                 -- Subsequent stages; sums of previous sums
109                                 prev_index := 0;
110                                 curr_index := num_sums;
111                                 num_sums   := num_sums/2;
112                                 for i in 1 to NSTAGES-1 loop
113                                         for j in 0 to num_sums-1 loop
114                                                 sum(curr_index + j) <= sum(
                                                        prev_index + 2*j) + sum(
                                                        prev_index + 2*j + 1);
115                                         end loop;
116
117                                         prev_index := curr_index;
118                                         curr_index := curr_index + num_sums
                                                ;
119                                         num_sums   := num_sums/2;
120                                 end loop;
121
122                 end if;
123                 end if;
124         end process;
125
126         -- ------------------------------------------------------------------
127         -- Full-precision output sum
128         -- ------------------------------------------------------------------
129         --
130         q <= std_logic_vector(resize(sum(NSUMS-1),OWIDTH));
131
132 end architecture;
```

Listing A.7 – Pacote utilizado para armazenar parâmetros do sistema

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.all;
4
5  package harris_package is
6
7
8          type kernel_type is array (natural range <>) of integer;
9
10         --Sobel
11         constant SOBEL_KERNEL_SIZE  : integer:=3;
12         constant SOBEL_KERNEL_DATA_WIDTH  : integer:=3;
13         constant SOBEL_MULT_WIDTH  : integer:=12;
14         constant SOBEL_OUTPUT_WIDTH  : integer:=12;
15         constant sobel_x_kernel : kernel_type := (1 ,0 ,-1, 2, 0, -2, 1,
               0, -1);
16         constant sobel_y_kernel : kernel_type := (-1, -2, -1, 0, 0, 0,1,
               2, 1 );
17
18         --Derivatives products
19         constant PRODUCT_SIZE  : integer:=15;
20
21         --Gauss
22         constant GAUSS_KERNEL_SIZE  : integer:=5;
23         constant GAUSS_KERNEL_DATA_WIDTH  : integer:=6;
24         constant GAUSS_MULT_WIDTH  : integer:=21;
25         constant GAUSS_OUTPUT_WIDTH  : integer:=30;
26         constant gauss_kernel :  kernel_type := (1, 2, 4, 2, 1, 2, 4, 8,
               4, 2, 4, 8, 16, 8, 4, 2, 4, 8, 4, 2, 1, 2, 4, 2, 1);
27
28
29 end harris_package;
```

Listing A.8 – Pacote utilizado com algumas funções adicionais

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.all;
4
5  package util is
6
7    function log2ceil (n : integer)
8      return integer;
9
10  function log2floor (n : integer)
11     return integer;
12
```

```
13   function FLOOR (
14      constant N : natural;                  -- Numerator
15      constant D : natural)                  -- Denominator
16      return natural;
17
18    function CEIL (
19      constant N : natural;                  -- Numerator
20      constant D : natural)                  -- Denominator
21      return natural;
22
23    function LOG2 (
24      constant i : natural)
25      return integer;
26  --
27
28  end util;
29
30  package body util is
31
32          -- purpose: computes ceil(log2(n))
33          function log2ceil (n : integer) return integer is
34
35          variable m, p : integer;
36          begin
37           m := 0;
38           p := 1;
39           for i in 0 to n loop
40                   if p < n then
41                           m := m + 1;
42                           p := p * 2;
43                   end if;
44           end loop;
45          return m;
46
47          end log2ceil;
48
49          -- purpose: computes floor(log2(n))
50          function log2floor (n : integer) return integer is
51
52          variable m, p : integer;
53          begin
54           m := -1;
55           p := 1;
56           for i in 0 to n loop
57            if p <= n then
58                           m := m + 1;
59                           p := p * 2;
```

```
60            end if;
61           end loop;
62          return m;
63
64          end log2floor;
65
66
67   -- purpose: Round towards minus infinity.
68    function FLOOR (
69      constant N : natural;                -- Numerator
70      constant D : natural)                -- Denominator
71      return natural is
72    begin
73      return N/D;
74    end FLOOR;
75
76    -- purpose: Round towards plus infinity.
77    function CEIL (
78      constant N : natural;                -- Numerator
79      constant D : natural)                -- Denominator
80      return natural is
81    begin
82      return (N-1)/D+1;
83    end CEIL;
84
85    -- purpose: Returns the binary logarithm
86    function LOG2 (
87      constant i : natural)
88      return integer is
89      variable tmp     : integer := i;
90      variable ret_val : integer := 0;
91    begin  -- LOG2
92      while tmp > 1 loop
93        ret_val := ret_val + 1;
94        tmp     := tmp / 2;
95      end loop;
96
97      return ret_val;
98    end LOG2;
99
100 end util;
```

ANEXO B – Código MATLAB

Listing B.1 – Algoritmo de detecção de cantos de Harris em MATLAB

```
function R = harris(I);

I = double(I);

[size_x size_y] = size(I);

sobel_x = [1 0 -1; 2 0 -2; 1 0 -1]
sobel_y = [-1 -2 -1; 0 0 0;1 2 1 ]

Ix = conv2(I,sobel_x,'valid');
Iy = conv2(I,sobel_y,'valid');


Ix2 = Ix.^2;
Iy2 = Iy.^2;
Ixy = Ix.*Iy;


gauss = [1 2 4 2 1; 2 4 8 4 2; 4 8 16 8 4; 2 4 8 4 2; 1 2 4 2 1 ]

Gx2 = conv2(Ix2,gauss,'valid');
Gy2 = conv2(Iy2,gauss,'valid');
Gxy = conv2(Ixy,gauss,'valid');

R = (Gx2 + Gy2) .^2;
R = floor(R / 32);
R = (Gx2.*Gy2 - Gxy.^2) - R;
```

Listing B.2 – Exemplo de teste de detecção de cantos de Harris, realizando uma limiarização e supressão não máxima

```
clear all; close all; clc


I=imread('lena512.bmp');


if(size(I, 3) > 1)
        I = rgb2gray(I);
end
```

```
10
11 R2 = harris(I);
12
13 %Theshold
14 r=3;
15 sze = 5*r+1;
16 MX = ordfilt2(R2,sze^2,ones(sze));
17 CR2 = (R2==MX) & (R2 > 0.005*max(max(R2)));
18
19 figure; imshow(CR2 , [])
20
21
22 I = I(4 : end - 3, 4 : end-3);
23 figure; imshow(I); hold on;
24 [i, j] = find(CR2 == 1);
25 for(k = 1:length(i))
26         plot(j(k), i(k), 'Xr', 'markersize', 10);
27 end
```